Руководство по программированию на Haxe

(для любопытных)

Содержание

Предполагаю, что читатель знает, что такое язык программирования Haxe, для чего он был создан и как его можно использовать. Тем, кто только что познакомился с самим этим названием Haxe, рекомендую предварительно прочитать хотя бы статью о языке в википедии. А здесь я, опуская всякие общие рассуждения, сразу начну с базовых понятий.

1.Первое знакомство (*helloworld*)

Итак, не нарушая традиций, познакомимся с программой ***helloworld***, которая на Haxe может выглядеть примерно так:

```
class Foo  {
        static function main()  {
                trace("Hello World!");
        }
}
```

Этот текст надо поместить в файл ***Foo.hx***, перейти в командной строке туда, где находится этот файл и скомпилировать его командой:

haxe -main Foo -neko hello.n

В той же директории появится файл ***hello.n***, который затем можно выполнить на виртуальной машине ***neko***, созданной разработчиками языка. Для этого надо выполнить команду:

neko hello.n

Получим следующий результат:

Foo.hx:3: Hello World!

Я думаю, что смысл всех элементов команд на компиляцию и на выполнение понятен и не требует дополнительных объяснений.

Имеется также такой вариант команды:

haxe -main Foo —inter

Теперь файл с расширением ***.n*** не будет создан, а программа будет сразу выполнена. Можно считать, что это интерактивный режим, удобный при отладке.

Значит, кроме приветствия программа выводит ещё и название исходного файла, а также номер строки в которой расположен оператор ***trace***, обеспечивающий вывод на терминал. Конечно, есть возможность вывода и другими способами. В частности, neko позволяет применять привычные операторы ***print*** и ***println***. Вставим в тело функции main такую строку:

neko.Lib.println("now Hello World!");

На экран будет выведено:

now Hello World!

Оператор ***trace*** удобно применять только на этапе отладки программы, в рабочем режиме нет смысла выводить название файла и номер строки. Можно отключить оператор ***trace***, применив в команде на компиляцию флаг ***no-traces***.

В библиотеке Нахе есть также класс ***Sys***, имеющий много статических методов (посмотрите файл ***Sys.hx*** в папке ***std***) среди которых есть методы ***print*** и ***println***. Так что вывод на терминал можно программировать и так:

Sys.println("now Hello World!");

Чтобы далее не возвращаться к этой теме, отмечу ещё, что при выводе можно использовать интерполяцию. Нахе выполняет интерполяцию при использовании одинарных кавычек, при двойных кавычках всё выводится «как есть». Перед интерполируемым выражением требуется поставить знак ***$*** и само выражение заключить в фигурные скобки (фактически это будет блок), одиночные переменные заключать в скобки не надо:

var x = 12;

Sys.println('Сумма $x и 3 равна ${x + 3}'); → Сумма 12 и 3 равна 15

Даже из текста этой рутинной программы можно сделать несколько существенных выводов. Во-первых, программа на Нахе должна иметь класс, содержащий функцию ***main*** без аргументов (пустые скобки обязательны), определяющую точку входа. Во-вторых, видим, что функции создаются директивой ***function*** (функция ***main*** обязательно со спецификатором ***static)***. Название файла должно совпадать с именем класса, содержащего функцию ***main***. Перед спецификатором ***static*** может быть ещё и спецификатор ***public***, но в данном контексте это не обязательно. Позже мы рассмотрим применение спецификаторов.

Компилятор Нахе позволяет получать файлы не только на специальном языке ***neko***, но также и на некоторых других языках, в частности, на ***java***. Чтобы получать программы на java надо

предварительно установить библиотеку ***hxjava***, подключившись к интернету и выполнив команду:

haxelib install hxjava

Команда на компиляцию в программу на java должна быть такой:

haxe -main Foo -java hello

Расширение у названия hello не требуется, так как это не название файла. В результате компиляции будет создана папка с названием ***hello***, содержащая огромное количество всяких файлов в нашем случае с общим объёмом в 279кб. Такое поведение характерно для вполне идиотского на мой взгляд языка java. В папке hello будет, в частности, файл ***Foo.jar***, который можно выполнить, как это обычно делается в системе java, то-есть надо перейти в папку hello и выполнить команду:

java -jar Foo.jar

Получим тот же самый результат.

Работу Нахе с другими языками пока не будем рассматривать, к этой теме мы ещё вернёмся впоследствии. Ну, а пока будем работать исключительно только с виртуальной машиной ***neko***.

Как и все другие современные языки, Нахе позволяет вводить аргументы из командной строки. Усложним программу, добавив в неё возможность ввода имени для приветствия, а также вычислим количество букв в этом имени:

```
class Name  {
        static function main()   {
                var name = Sys.args()[0];
                trace("Hello " + name);
                trace("Your name is  " + name.length + "
                characters long.");
        }
}
```

Поместим текст в файл ***Name.hx*** и скомпилируем программу командой:

haxe -main Name -neko hello.n

Команду на выполнение зададим в таком виде:

neko hello.n Anna

В результате получим:

Name.hx:4: Hello Anna
Name.hx:5: Your name is 4
characters long.

Аргументы, задаваемые после команды на выполнение, доступны в программе через функцию ***args*** из класса ***Sys***. Эта функция возвращает массив, элементы которого имеют тип ***String*** и принимают значения аргументов из командной строки. Элементы массива сразу же могут быть извлечены в обычной нотации с применением квадратных скобок. В программе объявлена переменная ***name*** со служебным словом ***var***, которая одновременно инициирована значением первого элемента массива (с индексом 0). Тип переменной не задан, и это значит, что тип (у нас String) выводится из контекста автоматически. Знак ***(+)*** в тексте, выводимом оператором trace, является знаком конкатенации. Встроенный метод ***length*** определяет количество букв в имени, приведение числа типа ***Int*** к типу ***String*** при конкатенации тоже выполняется автоматически. Текст для вывода вторым оператором trace задан в две строки, между двойными кавычками может быть несколько строк.

Нахе позволяет вводить в текст программы проверку некоторых условий, которые можно задавать в команде на компиляцию. Такой приём называют компиляцией с условиями. Условия в командной строке задаются с помощью ключа ***-D***. В самой программе применяется особый синтаксис. Лучше всего просто посмотреть на примере. Пусть в файле ***Main.hx*** имеется такой текст:

class Main {

```
    static function main() {
        #if Hello
            trace("Компиляция с условием.");
        #end
        #if (x > 4)
            trace("Привет, администратор!");
        #elseif (x > 2)
            trace("Привет, профессионал!");
        #else
            trace("Привет, чайник!");
        #end
    }
}
```

Значит, условные операторы ***if, else*** и ***elseif*** в этом специальном случае предваряются знаком ***#***, по которому компилятор их и распознаёт. Конец каждого условного выражения отмечается словом ***#end***. Сами условия, как уже сказано, задаются в команде на компиляцию, которая в данном примере должна выглядеть приблизительно так:

haxe -main Main -neko Main.n -D Hello -D x=3

То-есть после ключа ***-D*** может располагаться или какое-то значение, или может быть объявлена переменная, инициированная произвольным значением. Значения могут быть ***String***, ***Int*** или ***Float***. Тогда в тексте программы может проверяться или текст, или переменная.

В нашем примере мы получим следующий результат:

Main.hx:4: Компиляция с условием.
Main.hx:9: Привет, профессионал!

В команде на компиляцию могут применяться разнообразные флаги, полный список которых можно получить по команде:

haxe --help-defines

Разбираться здесь со всеми флагами нет смысла, но в дальнейшем мы может быть применим один-два из них.

Отмечу ещё, что комментарии на Нахе начинаются с двойного прямого слеша ***(//)***, а многострочный комментарий заключается между скобками ***/*...*/***. В примерах я не буду приводить комментарии, в небольших фрагментах программ они бессмысленны и только ухудшают восприятие текста. Кроме того, я сторонник кратких идентификаторов, применять имена со смыслом (так сказать, собственные имена) целесообразно только для долго живущих объектов и в программах, предназначенных для повторного использования. В примерах я почти всегда буду применять идентификаторы из одной-двух букв.

2.Базовый синтаксис

Типы

Нахе – язык со статической типизацией (проверка типов на этапе компиляции), но, как мы уже видели, также может выводить тип из контекста. Имеется обычный набор базовых типов: ***Int, Float, String, Bool.*** Регулярные выражения имеют тип ***Ereg***. Имеется также специальное значение ***null***, имеющее неопределённый тип ***Null*** . Значение null возвращается во всех случаях, когда результат неопределённый. Нет необходимости рассматривать базовые типы подробно, основные их свойства увидим по ходу дела. Отмечу только, что тип ***Int*** совместим с типом ***Float*** и везде, где требуется тип ***Float***, можно использовать также и ***Int***. Обратное - не верно.

При объявлении переменной её тип можно задать через двоеточие:
var x : Int = 77;
Соответственно, можно задать тип результата, возвращаемого функцией. Если результат отсутствует, указывается тип ***Void***:

function f(x:Float) : Void { … }

Слово ***var*** при объявлении переменной указывает только на то, что объявляется новая переменная. Если это слово опустить, будет ошибка. Если при объявлении переменной без инициации указать тип, то далее переменная может быть инициирована только значением заданного типа:

var s :String;
s = "London";

Можно объявить переменную и без указания типа и тогда тип будет выведен при инициализации:

var s;
s = "London";

С одним словом ***var*** можно объявить несколько переменных одного или даже разных типов (впрочем, не всегда):

var a, b : Bool, c : Int = 0;

Если переменную при объявлении инициировать значением ***null***, то потом ей можно присвоить значение любого типа:

var s = null;
s = 12.34;

Но после этого ***s*** уже будет иметь тип ***Float.***

В том случае, когда автоматическое преобразование типа не работает, тип можно изменить принудительно. Для этого в модуле ***Std*** имеются встроенные функции ***int*** и ***string***:

var x = 12.67;
var y = Std.int(x);
trace(y); → 12

(Здесь и всюду далее я буду использовать стрелку ***(→)*** вместо слов «получим», «будет равно» и так далее. При копировании кода для выполнения надо эти стрелки вместе с текстом убрать.) Переменная ***x*** имеет тип ***Float***, а переменная ***y*** получает тип ***Int***, при этом

округление не выполняется. В классе ***Math*** есть также метод ***floor***, выделяющий целую часть числа без округления:

trace(Math.floor(3.75)); → 3

Метод ***string*** позволяет число превратить в строку:

var x = 123.456;

var y = Std.string(x); → 123.456 и это строковое значение

Дальше в примерах мы будем иногда программировать ввод с клавиатуры, поэтому покажу здесь, как это делается. Ввод с клавиатуры выполняет функция ***readLine()*** из модуля ***Sys*** (вообще-то в модуле ***sys*** имеется только метод ***stdin***, вызывающий функцию ***readLine*** совсем из другого пакета). Чтобы инициировать переменную значением, введённым с клавиатуры можно сделать так:

var s = Sys.stdin().readLine();

Функция ***readLine()*** вводит строку после нажатия клавиши ***enter***. Таким образом, переменная ***s*** будет инициирована введённой строкой и получит тип ***String***. Если надо ввести число, следует тип String трансформировать с помощью встроенных функций ***parseInt*** и ***parseFloat*** из модуля ***Std***:

var x = "77";

var y = Std.parseInt(x); → переменная ***y*** имеет тип Int.

var x = "123.034";

var y = Std.parseFloat(x); → переменная ***y*** имеет тип Float.

Если вводится строка, состоящая из слов, разделённых пробелами, строку можно преобразовать в массив и далее работать с элементами массива. Но о массивах позже.

Ввести с клавиатуры одиночный символ позволяет функция ***getChar*** из модуля ***Sys***:

var c = Sys.getChar(true);

Параметр ***true*** указывает на то, что вводимый символ будет отражаться на экране. Если его заменить на ***false***, на экране мы ничего не увидим.

trace(c); → выводится цифровой код введённого с клавиатуры символа

Получить символ из цифрового кода позволяет функция ***fromCharCode*** из модуля ***String***:

var y = String.fromCharCode(112);

trace(y); → ***p***

Более подробно работу с текстом рассмотрим позже.

В модуле ***Math*** есть функция ***random***, позволяющая получить случайные числа типа ***Float*** в диапазоне от ***0*** до ***1***:

trace(Math.random()); → ***0.884468818393236***

Эта функция не принимает аргументов.

В модуле ***Std*** есть другая функция ***random***, позволяющая получать целые случайные числа (тип Int) из заданного диапазона:

trace(Std.random(100)); → случайное число из диапазона от ***0*** до ***100***, при этом ***0*** может быть получен, как результат, а ***100*** — нет.

Автономную единицу представляет блок — отрезок программного кода в фигурных скобках. Блок всегда возвращает последнее вычисленное значение:

var t = { var x = 1.5; Math.sin(x); }

neko.Lib.println(t); → ***0.997494986604054***

Переменная, объявленная в блоке, является локальной, она вне блока не видна и её нельзя объявить ***public***. Блоку можно присвоить имя, но это будет имя переменной, которая получит значение, возвращаемое блоком :

var y = 7;

var f = { var x = 3; x * y; }

var g = { var t = 4; f/t; }

trace(g); → ***5.25***

В модуле ***Type*** имеется функция ***typeof***, позволяющая узнать тип переменной:

trace(Type.typeof(x)); → будет выведен тип переменной ***x*** (с системой типов подробнее познакомимся далее).

Имеется также оператор ***$type***, который можно использовать для контроля типов при отладке программы. Этот оператор выводит сообщение о типе во время компиляции программы:

var x = 12.5;
$type(x); → Float

Нахе позволяет указывать сигнатуру функций с использованием общепринятого синтаксиса. Например, если сигнатура имеет вид ***:Int -> String -> Float***, то она соответствует функции, принимающей два аргумента типа ***Int*** и ***String*** и возвращающей результат типа ***Float***. Например мы можем объявить такую переменную:

var f :Float -> Float -> Float;

Это значит, что ***f*** – функция, принимающая два аргумента типа ***Float*** и возвращающая результат также типа ***Float***. Затем можно создать эту функцию:

```
function f(x, y) { return x*y; }
trace(f(3, 7)); → 21
```

Ясно, что если сигнатура не объявляется явно, то она выводится компилятором из контекста, как и любые другие типы. Оператор ***$type*** позволяет узнать сигнатуру функции:

```
class Prob1 {
    static public function main() {
        $type(f); → i : Int -> s : String -> Bool
        $type(f(1, "foo")); → Bool
    }
    static function f(i:Int, s:String):Bool { return true; }
}
```

Таким образом, если ***$type*** вызвать для функции без указания аргументов, то получим её сигнатуру, а если с аргументами — то получим только тип возвращаемого функцией результата. Если среди

аргументов будут необязательные (optional, смотрите далее), то в выводимой сигнатуре перед ними будет знак вопроса.

Можно применять аргументы «по умолчанию»:

```
class Prob  {
    public static function main() {
        trace(g()); → 25
        trace(g(7)); → 49
    }
    public static function g(x :Int = 5) {return x * x; }
}
```

Значит, аргументу можно задать значение прямо в списке аргументов и тогда при вызове функции параметр можно опустить, тогда его значение будет принято «по умолчанию», или указать новое значение.

Соответственно, допустимы и необязательные (optional) аргументы. Чтобы сделать аргумент optional, достаточно перед ним поставить знак вопроса:

```
function f(x :Float, ?y :String) {…}
```

При вызове функции необязательному аргументу можно не указывать значение и тогда он получит фиксированное значение ***null***. Конечно, «необязательность» надо отработать в теле функции. Соответствующий пример будет в разделе «ветвления».

Аргумент может быть одновременно «по умолчанию» и «optional»:

```
function f(?x :Int = -1) {…}
```

Впрочем, этот вариант мало чем отличается от просто по умолчанию. Ясно, что аргументы по умолчанию и optional должны находится в конце списка аргументов, если все они имеют один и тот же тип. При аргументах с разными типами нужный аргумент компилятор может узнать по типу.

Для выхода из функции до её окончания или для возвращения значения из функции используется выражение ***return*** . Оператор

return может использоваться без аргументов, если функция не должна возвращать значения.

Анонимные функции

Анонимные функции (обычно называют ***closure*** – замыкания) обязательный атрибут функциональных языков. Анонимные функции объявляются также, как и обычные, только им не присваивается идентификатор, а сразу после слова ***function*** следует список аргументов:

function(x, y) { return x+y; }

Такие функции в некоторых случаях могут использоваться напрямую, без присвоения им ненужного имени. Дальше мы увидим, насколько иногда это бывает полезно. Но имя всегда можно и присвоить, достаточно просто переменную инициировать этой анонимной функцией:

```
var f = function(x, y) { return x+y; }
trace(f(6, 7)); → 13
```

То-есть, переменная ***f*** при этом становится обычной функцией. Переменную можно объявить заранее и указать сигнатуру, если это зачем-то нужно:

```
var f :Int -> Int -> Int;
f = function(x, y) { return x+y; }
trace(f(6, 7));
```

В заключение этого общего обзора приведу ещё способы обмена информацией с файлами. Прочитать содержимое файла очень просто:

```
var s = sys.io.File.getContent( "Prob.hx" );
```

Метод ***getContent*** из класса ***File*** пакета ***sys.io*** в качестве аргумента принимает строку, представляющую путь для файла, относительный или абсолютный. В примере будет прочитано содержимое файла ***Prob.hx*** из текущей директории. Можно указать, также в кавычках, полный путь для файла. Переменная ***s*** будет иметь тип ***String***.

Так же просто выполняется запись в файл с помощью метода ***saveContent*** того же класса ***File***:

var s = "Не жалею, не зову, не плачу";
sys.io.File.saveContent("Main.hx", s);

Здесь в файл ***Main.hx*** текущей директории будет записано содержимое переменной ***s***. Старое содержимое файла будет утрачено. Если файл не существовал, он будет создан. Путь можно указать абсолютный. Конечно, есть возможность и записи в существующий файл без удаления имеющегося там содержимого.

Исключения

Основное назначение исключений - обработка ошибок времени выполнения. Можно выбросить (***throw***) исключение и поймать (***catch***) его из любой вызывающей функции в стеке :

function foo() { throw new Error("invalid foo"); }

try { foo(); } catch(e : Error) { обработка исключения }

Можно использовать несколько ***catch*** после ***try*** для того, чтобы отлавливать разные типы исключений. Они тестируются в порядке, в котором объявлены. Выражение ***catch(e : Dynamic)*** будет отлавливать все исключения :

try { foo(); } catch(e : String) { обработка ошибки}

catch(e : Error) {обработка других типов ошибок}

catch(e : Dynamic) {обработка всех остальных ошибок }

Все выражения ***try*** и ***catch*** должны иметь один возвращаемый тип, за исключением ситуаций, когда никакого значения не нужно (также как ***if***). Ограничусь только простейшим примером. Для Нахе деление на нуль — не аварийная ситуация, так как возвращается специальное значение ***1.#INF*** и программа не останавливается. Предусмотрим такую остановку с выводом текста ***«Деление на нуль»***:

```
class Prob2 {
	static function main() {
		try { f(0); } catch(e :String) {  trace(e); return; }
	}
	public static function f(x) {
		if (x == 0) { throw new String("Деление на нуль"); }
		else { trace(1/x); }
	}
}
```

В результате получим нужный текст, а оператор ***return*** обеспечит выход из программы. Если вызвать функцию ***f*** не с нулевым аргументом, например ***f(2);*** получим результат ***0.5***.

3.Классы

Нахе язык чисто объектно-ориентированный, программа содержит только классы (а также пару структур, похожих на классы) и ничего более. Как уже говорилось, должен быть класс, содержащий функцию ***main***, в которой, собственно, и находится код, управляющий ходом выполнения программы. Название головного модуля программы должно совпадать с названием класса, содержащим ***main***. Сама программа может состоять из нескольких модулей, расположенных в разных пакетах. Управление пакетами рассмотрим позже.

Конструктор

Начнём рассмотрение классов Нахе с конкретного примера — составим программу вычисления радиус-вектора точки на плоскости при заданных координатах точки ***x*** и ***y***. (Можно также считать, что вычисляется длина гипотенузы прямоугольного треугольника по заданным катетам):

```
class Prob   {
```

```
        static function main()  {
        var p = new Point();
        var d = p.dist(3.0, 4.0);
        neko.Lib.println("distance = " + d);  → distance = 5
        }
}
class Point  {
        public function new()  {  }
        public function dist(x:Float, y:Float)  {
        return Math.sqrt(x*x + y*y);
        }
}
```

Итак в программе ***Prob*** объявлен класс ***Point***, содержащий метод ***dist***, вычисляющий радиус-вектор. Метод объявляется с помощью ключевого слова ***function*** со спецификатором (модификатором) ***public***. Этот спецификатор обязателен, поскольку в Нахе по умолчанию все объявления считаются ***private*** и, значит, недоступны за пределами экземпляров класса. Сам спецификатор ***private*** можно писать, или не писать, как нравится. Функция ***new*** в классе является конструктором и всегда должна быть ***public***. В нашем случае эта функция не имеет ни аргумента, ни тела, но, тем не менее, она необходима для создания экземпляра класса.

Кстати, в других языках программирования по умолчанию обычно предполагается спецификатор ***public***, поскольку он встречается намного чаще, чем ***private***. Почему-то в Нахе приняли противоположный вариант, в результате чего в программном коде слишком часто присутствует слово ***public***, и это совсем не украшает код.

Экземпляр класса Point ***p*** создаётся в теле функции ***main*** с помощью оператора ***new***, скобки после имени класса обязательны.

Оператор ***new*** вызывает конструктор ***new()***, создающий экземпляр класса.

Вызов метода ***dist*** экземпляра класса ***p*** выполняется, как обычно, в точечной нотации:

p.dist(3.0, 4.0);

Возвращаемым методом ***dist*** результатом инициируется переменная ***d***, значение которой затем выводится на терминал.

Конструктор ***new()*** может иметь аргументы и выполнять заданные действия. В частности, наша программа может быть и в таком виде:

```
class Prob  {
      static function main()  {
      new Point(3, 4);  → Prob.hx: 8: Distance =  5
      }
}
class Point  {
      public function new(x:Float, y:Float)  {
            trace("Distance = " + Math.sqrt(Math.pow(x, 2) +
            Math.pow(y, 2)));
      }
}
```

Здесь мы всю работу поручили выполнить конструктору. Обратите внимание на то, что выражение, выполняющее вычисления, записано в две строки. При этом не требуется никакого знака переноса строки, надо только оборвать выражение на таком знаке, чтобы оборванная строка не имела смысла. У нас строка оборвана на знаке ***(+)***, что предполагает, что выражение должно иметь продолжение. Тело конструктора выполняется в момент создания экземпляра класса, при этом аргументы должны указываться в круглых скобках после названия класса.

Для того, чтобы в классе иметь доступ к аргументам, передаваемым конструктору при создании экземпляра, надо в конструкторе инициировать поля со словом ***this***:

```
class Prob  {
        static function main()  {
        var p = new A(1.2);
        trace(p.a);  → 1.2
        trace(p.f());  → 1.44
        }
}
class A  {
        public var a :Float;
        public function new(x :Float)  {
                this.a = x;
        }
        public function f() { return a * a; }
}
```

Здесь выражение ***this.a = x;*** инициирует поле ***a*** класса ***A*** значением, передаваемым классу при создании экземпляра ***p***. Поле ***a*** доступно для использования в классе ***A*** и экземпляр ***p*** тоже имеет доступ к полю ***a***. По общепринятой терминологии переменную ***a*** можно назвать переменной экземпляра класса. Впрочем, новый идентификатор ***a*** можно и не вводить и всегда поступают так:

```
class Prob  {
        static function main()  {
        var p = new A(1.2);
        trace(p.x);  → 1.2
        trace(p.f());  → 1.44
        }
}
class A  {
```

```
        public var x :Float;
        public function new(x :Float)  {
                this.x = x;
        }
        public function f() { return x * x; }
}
```

Надо только иметь ввиду, что здесь переменная экземпляра ***x*** и параметр ***x***, передаваемый конструктору — разные вещи.

Функции в классе могут вызывать друг друга через ***this***:

```
class Prob  {
        public static function main() {
        var p = new A();
        trace(p.f(5)); → 32
        }
}
class A  {
        public function new() {}
        public function f(x :Int) { return this.g(x) + 7; }
        public function g(y :Int) { return y * y; }

}
```

Впрочем, последние версии Нахе позволяют этот ***this*** опускать, хотя он, конечно, подразумевается.

Спецификатор *static*

Поля и методы могут иметь спецификатор ***static*** и тогда они будут полями и методами класса (а не экземпляра класса). Это значит, что теперь не требуется создавать экземпляр класса, а вызывать поля и методы можно напрямую, по имени самого класса. Раз не нужен экземпляр класса, то не нужен и конструктор ***new()***. При этом наша программа может выглядеть так:

```
import neko.Lib;
class Prob  {
      static function main()  {
      var d = Point.dist(3, 4);
      Lib.println("distance = " + d);  → distance = 5
      }
}
class Point  {
      public static function dist(x, y)  {
      return Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
      }
}
```

Строка ***import neko.Lib;*** импортирует модуль ***Lib*** и в дальнейшем вместо ***neko.Lib.println(...)*** можно писать ***Lib.println(…).*** Так программировать имеет смысл, когда программа много раз обращается к модулю ***Lib***, а оператор ***import*** загружает модуль один раз.

(То, что я в тексте по привычке называю модулем на самом деле часто является классом, имеющем статические методы, доступные по имени класса без создания экземпляра).

Свойства

Доступ к полям класса можно определять не только с помощью спецификаторов ***public*** и ***private***. Для этого Нахе позволяет также применять так называемые свойства, которые записываются в круглых скобках после идентификатора переменной. Всего указывается два свойства, первое определяет доступ по чтению, второе — по записи. Например:

var x(default, never) :Int;

Здесь объявлена переменная ***x*** типа ***Int*** для которой доступ по чтению определён словом ***default***, а по записи — словом ***never***. Допустимы следующие значения свойств:

- ***default*** – разрешает нормальный доступ, если при этом переменная имеет спецификатор ***public***.
- ***null*** – разрешает доступ только из того класса, где определена переменная (фактически это то же, что спецификатор ***private***).
- ***dynamic*** – доступ фактически не контролируется.
- ***never*** – доступ закрыт.
- ***get/set*** – должны быть созданы программистом соответствующие методы доступа.

Посмотрим применение некоторых из свойств на примере:

```
class Main {
        static function main() {
        var p = new A(123);
        trace(p.t); → 123
        p.set_t(555); trace(p.t); → 555
        p.t = 777; trace(p.t); → 777
        trace(p.s); → 777
        trace(p.get_s()); → 777
        }
}
class A {
        public var t(default, set_t):Int;
        public var s(get_s, null):Int;
        public function new(t:Int) {this.t = t;}
        public function set_t(v:Int) { t = v; return t; }
        public function get_s() {  return t; }
}
```

Применяется такой синтаксис: для переменной ***x*** методы должны именоваться ***get_x*** и ***set_x***, для переменной ***y*** – ***get_y*** и ***set_y*** и так

далее. Как видим, ***p.t = 55;*** и ***p.set_t(55);*** означают одно и то же. Если написано ***function get_s() { return t; }***, то переменная ***s*** будет равна переменной ***t***. В остальном технику работы со свойствами понять нетрудно по одному этому простому примеру.

Анонимные объекты

Анонимные объекты (их ещё называют анонимными структурами) можно создавать без использования классов, просто объявляя поля и методы в теле блока. При этом используется такой синтаксис:

var d = { a : 1, b : 2.3, c : true };
trace(d); → { a => 1, b => 2.3, c => true }

При выводе принята другая форма для объекта, во многих языках программирования такая форма используется для коллекций hash (или map, или отражение).

В нашем анонимном объекте имеется три поля ***a, b*** и ***c***. Значения полей извлекаются так:

trace(d.b); → 2.3

Поля объекта можно изменять:

d.a = 777;
trace(d); → { a => 777, b => 2.3, c => true }

При этом нельзя только изменить тип поля.

Идентификаторы полей (или ключи) всегда представлены строками, а значения могут иметь любой тип. Ещё пример:

```
function f(x)  { return Math.pow(x, 2); }
var c = 12;
var p = { name : "Anna", age : c, g : f };
neko.Lib.println(p.name); → Anna
neko.Lib.println(p.age); → 12
neko.Lib.println(p.g(5)); → 25
```

Следовательно, анонимный объект ***p*** похож на коллекцию, называемую обычно hash (или ассоциативный список), который

состоит из пар, в которых первый элемент называют ключ, а второй — значение. В данном случае в роли ключей - идентификаторы полей и методов, а значениями могут быть константы, переменные всех типов, выражения (блоки) и даже функции (у нас функция ***f***). Надо только иметь ввиду, что функции и переменные должны быть объявлены раньше объявления анонимного объекта. Вместо функции можно применить и блок, например, так:

```
class Prob  {
        static function main()  {
        var x :Float = 2.5;
        var p = { name :  "Anna", age : 12, g : { x * x; } };
        neko.Lib.println(p.g);  → 6.25
        }
}
```

Инициируя переменную анонимным объектом, мы присваиваем ему имя. Иногда такие объекты используют и без применения имени.

Конечно, анонимные объекты могут быть аргументами функций. Тип таких аргументов указывается, например так:

```
class Prob  {
        public function new() {}
        public function f(x :{name :String, age :Int}) {
                trace('Его зовут ' + x.name + ' ему ' + x.age + 'лет.');
        }
        public static function main() {
                var d = { age :25, name :  "Василий," };
                var p = new Prob();
                p.f(d);
        }
}
```

Результатом будет текст: ***Его зовут Василий, ему 25 лет***. Тип аргумента и в этом случае можно не указывать и тогда он будет

выведен компилятором, но работать это будет только в том случае, если все элементы анонимного объекта имеют один и тот же тип, например ***String***. Дело в том, что при автоматическом выводе типа все элементы считаются того же типа, как тип первого элемента.

При таком использовании анонимного объекта (или hash) аргументы функции становятся «именованными», поскольку при вызове они задаются с указанием имени. Достоинство такого способа в том, что передавать функции значения можно в любом порядке, и это бывает удобно, особенно если аргументов много.

В рассмотренном ранее примере анонимного объекта намного эффективнее было бы применить анонимную функцию:

```
var p = {name : "Anna", age:12,
    g : function(y :Float) { return y * y;} };
trace(p.g(7)); → 49
```

Анонимные объекты можно использовать и как hash, только ключи здесь должны быть представлены только идентификаторами (их можно записывать и в кавычках), а для извлечения значений применяется точечная нотация.

4.Ветвления и циклы

Ветвления: if

Нахе имеет условный оператор ***if-else***, который ничем не отличается от обычного, например:

```
class Prob  {
    static function main()  {
        neko.Lib.print("Введите возраст: ");
        var age = Std.parseInt(Sys.stdin().readLine());
        if(age<18)  {
            neko.Lib.println("Вы не взрослый.");
        } else  {
```

```
                neko.Lib.println("Вы взрослый");
            }
        }
}
```

(В примере я использовал кириллицу, если на вашем компьютере возникнут проблемы с её применением, используйте для текста программы код CP866).

Ветвь ***else*** может отсутствовать и тогда, если условие даёт ***false***, выполнение программы просто продолжается дальше.

Условный оператор возвращает результат и, значит, можно программировать так:

```
class Prob  {
    static function main()  {
        neko.Lib.print("Введите возраст: ");
        var age = Std.parseInt(Sys.stdin().readLine());
        var r = if(age<18)  { "Вы не взрослый."; }
                        else  { "Вы взрослый"; };
        neko.Lib.println(r);
    }
}
```

Или условный оператор можно вставит прямо в список для вывода:

```
neko.Lib.println(if(age<18)  { "Вы не взрослый."; }
                 else  { "Вы взрослый"; });
```

Конечно, после ***else*** может располагаться новая ветвь ***if-else:***

```
class Prob  {
    static function main()  {
        neko.Lib.print("Введите возраст: ");
        var age = Std.parseInt(Sys.stdin().readLine());
        if(age < 16)  {
            neko.Lib.println("Вы юноша!");
```

```
        } else if(age >= 16 && age < 18)  {
            neko.Lib.println("Вы почти взрослый");
        } else  {
            neko.Lib.println("Вы взрослый");
        }
    }
}
```

Здесь использована логическая операция ***(&&)*** - логическое ***И***. Соответственно, Нахе имеет логическое ***ИЛИ - (||)*** и логическое ***НЕ - (!)***.

Блоки после условного выражения и после ***else*** должны возвращать результат одного и того же типа. Если ветви ***else*** нет, оператор ***if*** возвращает результат типа ***Void***.

Допустима также краткая форма условного выражения с таким синтаксисом:

(условие)? выражение1 : выражение2;

Если условие даёт *true*, вычисляется *выражение1*, иначе - *выражение2*. Например:

```
var y :Float = 2;
var x = (y > 0 )? Math.log(y) : Math.log(-y);
trace(x);  →  0.693147180559945
```

(Кстати, при ***y = 0*** Нахе вернёт значение ***-1.#INF*** и программа не остановится)

Краткое условное выражение можно использовать в конструкторе для необязательного (*optional*) параметра экземпляра класса, например:

```
public function new( x : Int, ?y : Int )  {
    this.x = x;
    this.y = (y == null) ? 0 : y;
}
```

Перед необязательным параметром ставится знак вопроса, а в

конструкции ***this*** применяется условное выражение. Если при создании экземпляра задано значение только для ***x***, поле ***y*** будет иметь нулевое значение (а не ***null,*** как это было бы без условного выражения).

Ветвления: переключатель *switch- case*

Для выбора одного варианта из нескольких возможных удобнее применять переключатель ***switch- case***:

```
class Prob  {
      public static function main(): Void  {
            neko.Lib.print("Введите название города: ");
            var name = Sys.stdin().readLine();
                  switch(name)  {
                  case "Москва", 'Ленинград', 'Одесса' : neko.
                        Lib.println('Россия');
                  case 'Париж', 'Марсель' :
neko.Lib.println('Франция');
                  case 'Берлин' : neko.Lib.println('Германия');
                  default : neko.Lib.println('Я не знаю, что это за
страна');
                  }
      }
}
```

Как видим, один оператор ***case*** может принимать сразу несколько образцов для сравнения. Для этого надо перечислить их через запятую сразу после слова ***case***. Переключатель также возвращает результат, поэтому можно запрограммировать и так:

```
class Prob  {
      public static function main(): Void  {
            neko.Lib.print("Введите название города: ");
            var name = Sys.stdin().readLine();
```

```
            neko.Lib.println(
                switch(name)  {
                case "Москва", 'Ленинград', 'Одесса' : 'Россия';
                case 'Париж', 'Марсель' : 'Франция';
                case 'Берлин' : 'Германия';
                default : 'Я не знаю, что это за страна';
                } );
        }
}
```

Результат будет тот же.

В операторе ***case*** можно применять знакозаменитель (placeholder) ***(_),*** при его использовании строку ***default*** можно заменить на такую:
case _ : 'Я не знаю, что это за страна';

Здесь знак ***(_)*** означает «любой вариант», что совершенно равнозначно слову ***default***.

Есть и ещё один вариант ***case***, который можно использовать в том месте, где сопоставление должно быть удачным в любом случае. Заменим всё ту же строчку на такую:
case x : x;

Здсь ***x*** – локальная переменная, идентификатор её может быть любым. При сопоставлении эта переменная примет то значение, которое анализируется. То-есть, в нашем примере это будет то название города, которое мы введём с клавиатуры. Значит, если мы введём название города, отсутствующее в предыдущих ***case***, то это название и будет выведено на экран.

В функциональном программировании применение оператора ***switch-case*** называют «сопоставление с образцом» (***pattern matching***). Далее мы ещё отдельно рассмотрим некоторые возможности этого pattern matching в Haxe..

Циклы

Цикл ***while*** имеет две разновидности. В первом варианте условие проверяется до выполняемого блока:

```
var i : Int = 0;
    while(i < 5)  {
        trace(i); → 0,1,2,3,4
        i++;
    }
```

В данном случае если сразу при входе в цикл условие даёт ***false***, блок ни разу не будет выполнен.

Во втором варианте условие проверяется после блока, при этом дополнительно использовано служебное слово ***do***, указывающее начало цикла:

```
var i : Int = 0;
    do  {
        trace(i); → 0,1,2,3,4
        i++;
    } while(i < 5);
```

Теперь блок всегда будет выполнен хотя бы один раз. Так, если условие записать в виде ***(i > 5)***, получим одно значение ***0***.

Цикл ***for*** в Нахе всегда работает только с коллекциями:

```
class Prob  {
    public static function main(): Void  {
        for(i in 0...5)  {
            trace(i); → 0,1,2,3,4
        }
    }
}
```

Выражение ***(0...5)*** определяет коллекцию (обычно называют ***ранг***), элементы которой представлены числами ***0,1,2,3,4*** то-есть, правая граница в коллекцию не входит. На терминал ранг выводится в необычной форме:

```
var r = (0...5);
trace(r); → { max => 5, min => 0 }
```

В цикле ***for*** можно использовать коллекции разных видов:

```
class Prob  {
        public static function main(): Void  {
                var m = ["Люда", "Катя", "Саша"];
                for(i in m)  {
                        trace(i); → Люда, Катя, Саша
                }
        }
}
```

В этом примере в цикле ***for*** использован массив ***m***, элементы которого — значения типа ***String***. Ещё вернёмся к коллекциям позже.

Оператор ***break*** позволяет выходить из цикла досрочно по условию:

```
class Prob  {
        public static function main()  {
                for(i in 0...7)  {
                        if(i==4) { break; }
                trace(i); → 0,1,2,3
                }
        }
}
```

Оператор ***continue*** позволяет пропускать некоторые итерации в цикле по условию:

```
for(i in 0...7)  {
        if(i==4) { continue; }
trace(i); → 0,1,2,3,5,6
}
```

Обратите внимание — операторы ***break*** и ***continue*** записываются в фигурных скобках, как отдельный блок, в котором, конечно, можно выполнять и другие действия.

5.Коллекции

Массивы (Array)

Объявление массива на Нахе имеет такой синтаксис:

var m :Array<Int>;

То-есть, тип массива имеет вид ***:Array<Int>***, где ***<Int>*** указывает тип элементов массива. Затем массив можно инициировать списком в квадратных скобках:

m = [1, 2, 3, 4, 5];

Как обычно, тип можно и не указывать, и тогда при инициации он будет выведен из контекста. Элемент массива можно извлечь по индексу (он всегда имеет тип ***Int***) в квадратных скобках:

trace(m[2]); → 3

Элементы массива можно изменять (значит, массивы mutable):

m[3] = 7;

trace(m); → [1,2,3,7,5]

Элементы можно добавлять:

m[5] = 8;

trace(m); → [1,2,3,7,5,8]

Или так:

m[10] = 9;

trace(m); → [1,2,3,7,5,8,null,null,null,null,9]

Значит, пропущенные элементы получают значение ***null***. При извлечении элемента с несуществующим индексом тоже получим ***null:***

trace(m[25]); → null

При объявленном, а также и при не объявленном типе все элементы должны быть одного типа. И только, если объявить тип ***Dynamic***, в массиве могут быть элементы разных типов:

var m1 :Array<Dynamic> = [1.2, 'Hello', true];

trace(m1); → [1.2, Hello, true]

Массив ***m1*** имеет элементы типов ***Float, String*** и ***Bool***. Конкретнее о том, что означает задание типа в угловых скобках узнаем позже.

Поскольку Array класс, то создавать массивы можно с использованием конструктора класса:

var m2 = new Array<Dynamic>(); → так создаётся пустой массив

m2 = [m, m1];

trace(m2); → [[1,2,3,7,5,8,null,null,null,null,9],[1.2,Hello,true]]

Элементы массива ***m2*** представлены массивами, то-есть, ***m2*** это двумерная матрица. Для извлечения элемента из неё требуется двойной индекс:

trace(m2[1][2]); → true

Пустой массив можно создать и так:

var m = []; → его можно инициировать элементами любого, но только одного типа

Или, с указанием типа:

var m :Array<Int> = [];

Массив можно наполнить элементами (инициировать) с помощью цикла ***for***:

var m:Array<Float> = [for (x in 0...5) x];

trace(m);→ [0,1,2,3,4]

В цикле можно инициировать массив экземплярами класса:

```
class Main {
    static function main() {
        var s:Array<Array<Float>> = [[2,4], [8,3], [0,6]];
        var m:Array<Point> = [for (x in s) new Point(x)];
        trace(m); →[{ p => [2,4] },{ p => [8,3] },{ p => [0,6] }]
```

```
    }
}
class Point {
    public var p:Array<Float>;
    public function new(p) { this.p=p; }
}
```

В классе ***Array*** имеется традиционный набор методов для работы с массивами: ***length, concat, join, pop, push, reverse, shift, slice, sort, insert, remove, copy, map, filter*** и некоторые другие:

var m = [1,2,3,4,5];

m.reverse(); → метод изменяет порядок элементов «на месте»

trace(m); → [5,4,3,2,1]

trace(m.length); → 5

Встроенный метод ***split*** позволяет строку разбивать на части по заданному знаку (по разделителю) и преобразовывать её в массив:

var s = "Жили были дед да баба";

var m = s.split(" "); → в качестве разделителя пробел

trace(m); → [Жили,были,дед,да,баба]

trace(m[2]); → дед

Разделитель указывается, как аргумент метода ***split***. Разделителем может быть любой знак, например буква ***д***:

var m = s.split("д");

trace(m); → [Жили были ,е, ,а баба] → сам разделитель исчезает

или запятая:

s = "казнить, нельзя помиловать";

var m = s.split(",");

trace(m); → [казнить, нельзя помиловать]

trace(m[1]); →нельзя помиловать

В качестве разделителя можно задавать группу знаков:

trace(s.split("ель")); → [казнить, н,зя помиловать]

trace(s.split("ель")[0]); →казнить, н

Списки (List)

Списки можно объявлять также, как массивы, заменив только слово ***Array*** на ***List***:

var a :List<Dynamic>;

Этот способ позволяет только объявить список, но не создаёт пустого списка и объявленный так список нельзя инициировать с помощью знака равенства какой-нибудь коллекцией, кроме другого списка. Получить пустой список можно так:

var a = new List<Dynamic>(); → a – пустой список

Теперь к списку ***a*** можно применять методы из класса ***List***, например, метод ***add***, добавляющий элемент в конец списка:

a.add('Peter');
a.add(12.55);
trace(a); → {Peter, 12.55}

Как видим, элементы списков заключены в фигурные скобки. Нет возможности извлекать элементы списка по индексу.

Пустой список можно также создать с помощью метода ***createEmptyInstance*** из модуля ***Type***:

var a :List<Int> = Type.createEmptyInstance(List);
trace(a); → {}

Список можно инициировать другой коллекцией, например, массивом, с помощью функции ***list*** из модуля ***Lambda***:

var a :List<String> = Lambda.list(["Москва", "Ростов", "Вологда"]);
trace(a); → {Москва, Ростов, Вологда}

Список можно использовать в цикле ***for*** (то-есть, списки iterable)***:***

for (x in a) { trace(x); }

В классе ***List*** имеется набор методов для работы со списками, многие совпадают с соответствующими методами для массивов: ***length, push, first, last, pop, isEmpty, clear, remove, toString, join, filter, map*** и другие. Ещё есть ряд методов для списков в модуле ***Lambda***.

Векторы (vector)

Векторы в Haxe почти полностью подобны массивам. Главное их отличие в том, что вектор имеет фиксированный размер (длину), определяемый при создании вектора и в дальнейшем не может меняться. Для объявления вектора надо использовать стандартный класс ***Vector*** из модуля ***ds***:

```
class Prob {
    static function main() {
        var v = new haxe.ds.Vector(10);
        for (i in 0...v.length) {
            v[i] = i;
        }
        trace(v[0]); → 0
        trace(v[5]); → 5
        v[15] = 33; → Компилятор ошибку не фиксирует, но сам вектор не изменяется
        trace(v); → [0,1,2,3,4,5,6,7,8,9]
    }
}
```

Вектор нельзя инициировать, например так:

```
v = [1,2,3,4,5];
```

Map

Кроме анонимных объектов (структур), во многом похожих на отображения, Haxe позволяет использовать и настоящие Map (отображения), которые практически ничем не отличаются от подобных структур в других языках. На следующем примере показаны основные действия с ***Map***:

```
class Prob {
    static public function main() {
```

```
        var m1:Map<Int, String> = [1 => "one", 2 =>"two"];
        var m2 = ["one"=>1, "two"=>2, "three"=>3];
        $type(m2); → Map<String, Int>
        var m4 = ["M"=>"Monday", "T"=>"Tuesday"];
        trace(m4["M"]); →  Monday
        for (x in m4) { trace(x); } →Monday Tuesday
        for (x in m4.keys()) { trace(x); } → M T
        var m5 = [ for (x in m4.keys()) x => "FRIDAY!!" ];
        trace(m5); → {M => FRIDAY!!, T => FRIDAY!!}
        var m:Map<String, Any> = ["a" => 1, "b" => true, "c" =>
            'Hello'];
        trace(m); →  {a => 1, c => Hello, b => true}
    }
}
```

Ключи могут иметь любой тип, но одинаковый для всех элементов Map. Значения могут быть разных типов в одном Map, если указать для них тип ***Any***, или ***Dynamic***. По умолчанию цикл ***for*** работает со значениями, но можно использовать и ключи, воспользовавшись встроенной функцией ***keys***. Для извлечения значения указывается ключ в квадратных скобках.

В классе ***Reflec***t есть метод ***fields***, который позволяет выбрать значения из отображения в формате массива:

```
var s = {name : "Anna", age : 25, address : "Paris" };
var m = Reflect.fields( s );
trace(m); →  [name, age, address]
```

Сортировка коллекций

Метод ***sort*** позволяет сортировать массивы. В качестве аргумента он принимает функцию, которая сравнивает два объекта и возвращает ***1***, если условие даёт ***true*** и ***-1*** в противном случае. Для примера рассмотрим сортировку слов в заданном тексте:

```
class Main {
	static function main() {
		var s = "Белеет парус одинокий в тумане моря голубом";
		var m:Array<String> = s.split(" ");
		m.sort(function(a, b) return a > b ? 1 : -1);
		trace(m);
	}
}
```

Здесь методу ***sort*** передаётся анонимная функция. Заданный текст сначала преобразуем в массив с помощью метода ***split***, разделяя текст на слова по пробелам. Полученный массив затем сортируется по возрастанию. В результате получим следующий массив:
[Белеет,в,голубом,моря,одинокий,парус,тумане]

Если в условии заменить знак ***(>)*** на ***(<)***, массив будет сортироваться по убыванию и тогда получим:
[тумане,парус,одинокий,моря,голубом,в,Белеет]

6.Система типов Haxe

Явное и скрытое назначение типа

В предыдущих примерах мы уже познакомились с использованием базовых типов и видели, что тип можно назначить явно при объявлении переменной, или тип может быть получен неявно при инициализации переменной конкретным значением. И в том и в другом случае переменной, получившей тип, нельзя присвоить значение другого типа. Можно, конечно, переменную объявить заново с тем же именем, присвоив ей новый тип. Это будет работать, однако без каких-то веских причин так поступать не стоит, поскольку такой код всегда читается труднее.

В Haxe любой объект имеет какой-то тип и мы имеем возможность создавать свои собственные типы. В частности, каждый созданный

нами класс определяет новый тип. Кроме классов типы создают также интерфейсы (interface), с которыми познакомимся чуть позже. Собственные типы можно использовать точно также, как и базовые. Так, если мы имеем класс ***A***, то мы можем объявить переменную, присвоив ей этот тип ***A***, явно или неявно. Далее переменную типа ***A*** можно инициировать конкретным объектом класса ***A***, создав этот объект с помощью конструктора ***new***:

```
class Foo  {
        static function main()  {
        var x :A;
        x = new A();
        x.f();  → Hello
        }
}
class A  {
        public function new()  { }
        public function f()  {
                trace("Hello");
        }
}
```

Наследование и объекты, имеющие не единственный тип

Нахе имеет обычный механизм наследования. Покажем на примере, как это может выглядеть конкретно:

```
class Prob  {
        static function main()   {
                var p :B;
                p = new B();
                Sys.println(p.g(22));  → 44
                Sys.println(p.x);  → 77
                Sys.println(p.f(7));  → 49
```

```
            Sys.println(p.a);          → Hello
        }
}
class A  {
        public var a :String;
        public function new(a)  { this.a = a; }
        public var x = 77;
        public function f(y)  { return y*y; }
}
class B extends A  {
        public function new()  { super("Hello"); }
        public function g(b)  { return 2*b; }
}
```

Значит, слово ***extends*** указывает, что класс ***B*** (дочерний) наследует классу ***A*** (родительский, или супер-класс). Кроме того, в конструкторе дочернего класса должна быть директива ***super***, вызывающая конструктор ***new*** родительского класса. Директива ***super*** может принимать аргументы, передаваемые конструктору. Как видим, все поля и методы класса ***A*** доступны для экземпляра класса ***B***.

Механизм наследования Haxe позволяет создавать объекты, имеющие не один, а несколько типов. Такая возможность выглядит несколько необычно и отсутствует в других языках программирования. В предыдущем примере можно при объявлении переменной ***p*** указать тип ***A***:

var p :A;

а далее инициировать её экземпляром класса B:

p = new B();

и при этом всё будет работать, недоступным будет только метод ***g(b)*** из класса ***B***, поскольку он не имеет никакого отношения к типу ***A***. Следовательно, для переменной ***p*** мы имеем двойной тип ***A*** и ***B***. Двойной тип удобен тем, что такой объект можно использовать везде,

где требуется любой из этих типов. Ясно, что типов у одного объекта может быть и больше двух. Рассмотрим ещё один, более содержательный пример:

```
class Main  {
        public static function main()  {
                var x : Human;
                x = new Human("Владимир");
                f(x);  → Сидоров Владимир
        }
        public static function f(y : Liv)  { trace("Сидоров " + y.name); }
}
class Liv  {
        public var name : String;
        public function new()  { name = "Иван"; }
}
class Human extends Liv  {
        public function new(name : String)  {  super(); this.name = name; }
}
```

Функция ***f*** имеет аргумент типа ***Liv***, а мы передаём ей переменную ***x***, представляющую экземпляр класса ***Human***, имеющую тип ***Human***. То-есть, хотя переменная ***y*** имеет тип ***Liv***, ***y.name*** возвращает значение ***Владимир***, а не ***Иван***. Значит, переменная ***x*** имеет два типа одновременно: тип ***Human*** и тип ***Liv***. Если объявить переменную ***x*** сразу с типом ***Liv***:

```
var x : Liv;
```

всё будет работать также, поскольку далее мы инициируем её значением типа ***Human***:

```
x = new Human("Владимир");
```

Кстати, экземпляр класса ***Liv*** создаётся оператором ***super()***, при этом имя экземпляра не используется, а само его создание происходит как бы скрытно от нас.

Мы ещё увидим впоследствии, какую пользу можно иметь от возможности создания переменных множественного типа.

Интерфейсы (interface)

Кроме классов, типы могут создавать интерфейсы. Они похожи на классы, только вместо слова ***class*** применяется слово ***interface***. В теле интерфейса могут только объявляться поля и методы без их определения. Классы могут некоторым образом наследовать интерфейсам и для этого применяется служебное слово ***implements***, вместо ***extends.*** Посмотрим на простой пример:

```
class Prob  {
        public static function main()  {
                var p :A;
                p = new A();
                q(p);  → Привет Мир!
                trace(p.a);  → 0.777
        }
        public static function q(x :B)  { x.g("Мир!"); }
}
class A implements B  {
        public function new()  {  }
        public function g(x) { trace("Привет " + x); }
        public var a = 0.777;
}
interface B  {
        public function g(s :String) : Void;
        public var a :Float;
}
```

В интерфейсе ***B*** объявлены метод ***g*** и поле ***a***. При этом задан тип аргумента ***s*** метода ***g***, тип возвращаемого методом результата, а также

тип поля ***a***. Само определение метода и поля находится в классе ***A***. При этом в классе уже нельзя изменить ни сигнатуру метода, ни тип поля.

Созданный экземпляр ***p*** класса ***A*** имеет двойной тип ***A*** и ***B***. В этом легко убедиться, если изменить объявление переменной ***p*** на

var p :B;

При этом всё будет работать также.

Кстати, все поля и методы, объявленные в интерфейсах по умолчанию всегда ***public***, поэтому в нашем примере этот спецификатор можно было опустить.

Отметим ещё, что интерфейсы не могут создавать экземпляры и интерфейсы могут ***implements*** другие интерфейсы.

Однако, пока не очень ясно, какая польза от этих интерфейсов, посмотрим, что будет дальше.

Функции, как аргументы и как результаты других функций

Как и все прочие объекты, Функции Нахе имеют тип, который ещё иначе называют сигнатурой. Поэтому они могут быть аргументами других функций, а также функции могут возвращать результат, представляющий другую функцию. Посмотрим на такой пример:

```
class Name  {
      static function main()   {
      var m :Array<Int> = [3, 5, 8, 2, 9, 4, 6];
      var m1 :Array<Int> = m.filter(f);
      Sys.println(m1);  → [8, 2, 4, 6]
      var m2 :Array<Dynamic> = [4, "Самара", true, "город",
"река"];
      var m3 = m2.filter(function(x) { return Std.is(x, String); });
      Sys.println(m3);          → [Самара, город, река]
      }
      static function f(x :Int) :Bool {
```

```
            if (x % 2 == 0) { return true; } else { return false; }
        }
}
```

Здесь использован метод ***filter*** из класса Array, который принимает функцию в качестве аргумента. При первом применении метод ***filter*** имеет такую сигнатуру:

filter :(Int->Bool)->Array<Int>

То-есть, метод принимает функцию, имеющую аргумент типа ***Int*** и возвращающую результат типа ***Bool***. Сам метод возвращает результат типа ***Array<Int>***. Метод ***filter*** передаёт принятой функции элементы массива по одному и если функция после анализа элемента возвращает ***true***, добавляет элемент в массив — результат. В примере отбираются чётные числа. При втором применении метод ***filter*** имеет несколько другую сигнатуру:

filter :(Dynamic->Bool)->Array<Dynamic>

В этом случае отбираются элементы, имеющие тип ***String***.

При первом применении функцию-аргумент ***f*** мы определили отдельно и дали ей имя, а во втором случае использована анонимная функция, которая определена непосредственно при передаче аргумента методу.

Придумаем теперь хотя бы примитивную функцию, возвращающую другую функцию:

```
class Name  {
        static function main()   {
        var g = func();
        Sys.println(g(2, 3));  → 5
        }
        static function func() { return f; }
        static function f(x :Float, y :Float) :Float { return x + y; }
}
```

Функция ***func*** имеет такую сигнатуру:

func() :(Float -> Float -> Float)

То-есть, ***func*** не имеет аргументов и возвращает функцию.

Ещё один пример на эту тему:

```
class Prob  {
    public static function main() {
        var Some = function(a:Int, b:String):String  {
            return b + (++a);
        }
        var Other = function(f:Int->String->String):Void  {
            trace(f(10, "res: "));
        }
        Other(Some);  →res: 11
        }
}
```

Предлагаю разобраться, почему выводится (***res: 11),*** а не (***11 res:).***

Динамические (dynamic) функции

Функцию можно объявить с модификатором ***dynamic*** и тогда её можно изменять по ходу дела:

```
class Name  {
    public function new() {  }
    public static function main(): Void  {
        var m = new Name();
        m.f();  → Hello.
        m.f = function() { trace("Привет."); };
        m.f();  → Привет.
    }
    public dynamic function f() :Void { trace("Hello."); }
}
```

Для того, чтобы модифицировать динамическую функцию, достаточно её просто инициировать другой анонимной функцией.

При этом нельзя изменить сигнатуру функции. Значит, в нашем примере новая функция не должна иметь аргументов и ничего не должна возвращать.

В этом примере использована также возможность создания экземпляра самого головного класса и для этого надо в нём предусмотреть конструктор ***new***. Дальше я буду чаще применять этот приём. Кстати, если в головном классе нет конструктора, то не только функция ***new***, но также и все другие функции в этом классе (функции класса) должны объявляться со спецификатором ***static***, иначе они будут недоступны.

Для того, чтобы отметить ещё одно важное обстоятельство, напишем предыдущую программу в несколько другом виде:

```
class Name  {
        public function new() { }
        public static function main(): Void  {
                var m = new Name();
                var n = new Name();
                m.f();  → Hello.
                m.f = function() { trace("Привет."); };
                m.f();  → Привет.
                n.f();  → Hello.
        }
        public dynamic function f() { trace("Hello."); }
}
```

Здесь мы сделали два экземпляра класса ***Name***. Затем, в экземпляре ***m*** мы модифицировали динамическую функцию ***f***. При этом во втором экземпляре ***n*** функция осталась не модифицированной. Значит, модификация происходит на уровне экземпляра, а не на уровне класса.

Статические функции тоже можно модифицировать:

```
class Name  {
```

```
    public static function main(): Void  {
        f(); → Hello.
        f = function() { trace("Привет!"); };
        f(); → Привет!
    }
    public static dynamic function f() { trace("Hello."); }
}
```

Duck typing

Кто-то уже давно придумал эту метафору: *«если ходит и крякает, как утка, то это и на самом деле утка»*. Метафору используют для характеристики такого приёма программирования, при котором тип объекта не задаётся в жёсткой форме. Проще всего понять суть такого подхода на примере:

```
class Prob  {
    public var name : String;
    public function new(name : String, age :Int) { this.name = name; }
    public static function main(): Void {
        f(new Prob("Виктор!", 25));  →  Привет, Виктор!
    }
    public static function f(p : {name : String}) { trace("Привет, " +
p.name); }
}
```

Здесь аргумент функции ***f*** представлен объектом ***p***, имеющим поле ***name*** тапа ***String***. Такой синтаксис с указанием полей в фигурных скобках мы использовали ранее для анонимных объектов. Никакой другой информации об объекте ***p*** нет и это значит, что функция ***f*** может принимать любой объект с таким полем. В примере функции ***f*** передаётся экземпляр класса ***Prob.*** На самом деле этот экземпляр имеет два поля: ***name*** и ***age***, но функция ***f*** использует только одно поле ***name*** и её не интересует наличие второго поля. Можно считать,

что такой подход ориентирован больше на поведение объекта, чем на анализ его содержания.

Давайте теперь немного отдохнём от теории и просто посмотрим на ещё один пример:

```
class Prob  {
	public static var a :List<Int>;
	public static function main() {
		a = Lambda.list([1, 2, 3, 4, 5]);
		f(fi); → 12345
	}
	public static function fi(i :Int) { Sys.print(i); }
	public static function f(fun :(Int->Void)) : Void {
		for(i in a) { fun(i); }
	}
}
```

Итак, мы познакомились немного с системой типов Haxe. Однако, это пока только верхушка айсберга, попробуем теперь заглянуть поглубже.

7.Параметры типа (type Parameters) и динамическое программирование

Параметры типа

Ранее мы использовали объявление, например, массива, в такой форме:

```
var m :Array<Float>;
```

Указатель типа в угловых скобках задаёт тип для класса Array, определяя тип элементов массива. Обычно такой указатель для классов называют параметром типа. Соответственно, аналогичный параметр типа можно применять к любому классу. Для массива мы

указали конкретное значение параметра типа, а в общем случае применяется абстрактный тип. Для его обозначения можно использовать любую букву в верхнем регистре, но обычно по традиции применяют букву ***T.*** Можно считать, что ***T*** соответствует произвольному типу. Рассмотрим пример:

```
class Name  {
        public static function main(): Void {
        var p = new A("aa", "bb", "cc");
        trace(p.f());  → [aa,bb,cc]
        trace(p.x + p.y);  → aabb
        }
}
class A<T> {
        public var x :T; public var y :T; public var z :T;
        public function new(x, y, z) {this.x = x; this.y = y; this.z = z; }
        public function f()  { return [x, y, z]; }
}
```

Класс ***A*** принимает три аргумента и имеет один метод ***f***, возвращающий переменные экземпляра в массиве. Параметр типа ***T*** получает конкретное значение при создании экземпляра ***p*** класса. В примере тип выводится из контекста и получает значение ***String***. Если написать:

```
var p = new A(1, 2, 3);
```

то ***T*** примет значение ***Int***.

Таким образом, применив абстрактный тип, мы получили возможность использовать любые типы. При этом все элементы массива должны иметь одинаковый тип. Конечно, тип можно задать и конкретно, например:

```
var p = new A<String>("aa", "bb", "cc");
```

Классы с параметром типа допускают применение разнообразных трюков. Вот один из примеров:

```
class Main {
public static function main () {
        var x:A<Int> = A.f();
        x.v = 10;
        trace(x.v); → 10
        var y:A<Array<String>> = A.f();
        y.v = ['один', 'два', 'три'];
        trace(y.v); → [один,два,три]
        }
}
class A<T> {
        public static function f<T>():A<T> { return new A<T>(); }
        public var v:T;
        private function new () {}
}
```

Статический метод ***f*** класса ***A*** возвращает функцию-конструктор этого же класса. При вызове функции ***f*** выполняется приватный конструктор класса ***A***, при этом создаваемым экземплярам класса (***x*** и ***y***) можно задавать разный тип. В примере экземпляр ***x*** имеет тип ***Int***, а экземпляр ***y*** – тип ***Array<String>***. Впрочем, тот же результат можно получить с помощью типа Dynamic.

Параметру типа класса можно задать дополнительно ограничение с таким синтаксисом:

```
class A<T :(B, C)>
```

Такое ограничение означает, что объекты с типом ***T*** должны иметь обязательно двойной тип ***B*** и C. Изменим предыдущую программу:

```
class Main  {
        public static function main(): Void {
        var p = new A(11, 20, 15);
        trace(p.f());
        trace(p.x + p.y);
```

```
        }
}
class A<T :(Int, Float)> {
        public var x :T; public var y :T; public var z :T;
        public function new(x, y, z) {this.x = x; this.y = y; this.z = z; }
        public function f()  { return [x, y, z]; }
}
```

Теперь аргументы класса ***A*** должны одновременно представлять типы ***Int*** и ***Float***. В Нахе это только числа без десятичной точки. Если написать:

```
var p = new A(0.5, 2.44, 7.01);
```

то это не пройдёт, так как эти числа не могут представлять тип ***Int***.

Конечно, при использовании только базовых типов всё это не слишком интересно. Но посмотрим теперь на такой пример:

```
class Name  {
        public static function main(): Void {
        var p1 = new A(5);
        var p2 = new A(3);
        var p = new D(p1);
        p.fi();  → 25
        p.gi();  → 125
        var q = new D(p2);
        q.fi();  → 9
        }
}
class A implements B implements C {
        public var x :Float;
        public function new(x) {this.x = x; }
        public function f()  { return x * x; }
        public function g()  { return Math.pow(x, 3); }
}
```

```
interface B { public function f() :Float; }
interface C { public function g() :Float;}
class D<T :(B, C)> {
        public var x :T;
        public function new(x) {this.x = x; }
        public function fi() {trace(x.f());}
        public function gi() {trace(x.g());}
}
```

При использовании интерфейсов объекты ***p1*** и ***p2*** имеют двойной тип ***B*** и ***C*** (точнее даже тройной тип ***A, B, C***). Параметр типа класса ***D*** задан с ограничением ***D<T :(B, C)>*** и, значит, переменная экземпляра **x** должна представлять типы ***B*** и ***C*** и не может иметь никаких других вариантов. Следовательно, аргументами класса ***D*** могут быть объекты ***p1*** и ***p2***. Методы ***fi*** и ***gi*** используют эти объекты для вызова функций ***f*** и ***g***. Можно отметить, что применение параметра типа позволило нам в этом примере создать довольно жёсткие ограничения на типы, иногда это полезно.

Haxe позволяет применять так называемые ***extern***-классы, позволяющие реализовать некоторые дополнительные возможности. Такие классы объявляются со спецификатором ***extern***. В библиотеке Haxe есть множество таких классов. Например, класс ***EitherType*** позволяет создавать функции, способные принимать аргументы любого типа, указанного в списке типов. Например:

```
import haxe.extern.EitherType ;
class Prob {
        static public function main() {
                f(77) → 77
                f('Hello'); → Hello
                f(true); → будет ошибка
        }
        static function f(x :EitherType<Int, String>) { trace(x);}
```

}

Перед использованием класс ***EitherType*** надо импортировать. Переменной, которая должна принимать значения указанных типов надо задать тип ***EitherType*** с ограничением, в котором перечислить через запятую все нужные типы. В нашем примере функция ***f*** может принимать типы ***Int*** и ***String***, и никакие другие. Кстати, в extern-классах все поля и методы по умолчанию ***public,*** а спецификатор ***private*** надо указывать явно.

Динамический тип

Ранее мы уже использовали тип ***Dynamic***, теперь познакомимся с ним более детально. Фактически этот тип позволяет как бы отстраниться от системы типов Нахе, поскольку в этом случае компилятор уже не контролирует тип переменных. Когда переменная объявляется с типом ***Dynamic***, ей можно присваивать любое значение:

```
var x :Dynamic;
x = "Привет";
x = {one : 1, two : 2};
x = new Array<String>();
```

и так далее. Можно считать, что ***x*** не имеет никакого типа. Допустимо поступать, например, и так:

```
var x :Dynamic;
var y :Int;
x = "Привет";
y = x;
```

Здесь переменная ***y*** получит текстовое значение ***"Привет"***, хотя ранее она была объявлена, как ***Int***. (Тем не менее нельзя далее написать, например, ***y = true;*** объявление ***var y :Int;*** остаётся в силе). Надо, правда, заметить, что авторы языка предупреждают о необходимости подобные вещи делать осторожно, поскольку иногда это приводит к непредсказуемому поведению.

Переменные типа ***Dynamic*** сами могут иметь сколько угодно полей, всегда тоже типа ***Dynamic***. При этом используется такой синтаксис:

var h :Dynamic;
h = { }; → пустой hash
h.name = "Людмила";
h.age = 21;
trace(h); → { name => Людмила, age => 21 }

То-есть, в результате получаем ***hash*** (или анонимный объект). Полям далее можно присваивать значения любого типа, например:

h.age = false;
trace(h); → { name => Людмила, age => false }

Динамические переменные можно инициировать функциями и далее выполнять их:

```
var f :Dynamic;
f = function(x :Float) { return Math.pow(x, 3); }
trace(f(2.5)); → 15.625
```

Или оформить, как метод какого-то объекта:

```
var p :Dynamic = List;
p.f = function(x :Float) { return Math.pow(x, 3); }
trace(p.f(2.5)); → 15.625
```

Переменную ***p*** надо инициировать каким-нибудь доступным классом, всё равно каким. Здесь использован класс ***List***. Можно класс заменить его экземпляром, например, вместо ***List*** можно было бы поставить просто пустые скобки ***[]***, или ***{}***.

Тип ***Dynamic*** представляет класс ***Dynamic***, а это значит, что этот класс можно параметризировать. В этом случае все поля динамической переменной получат указанный тип:

```
var s :Dynamic<String>;
s.name = "Юрий";
s.age = 18; → будет ошибка, так как все поля теперь имеют тип String.
```

В модуле ***haxe*** есть специальный тип ***DynamicAccess***, который по существу — абстрактный тип, позволяющий работать с анонимными объектами (структурами), с использованием ключей типа ***String***:

```
class Prob1 {
    static public function main() {
        var user:haxe.DynamicAccess<Dynamic> = {};
        user.set("имя", "Шурик");
        user.set("age", 25);
        trace(user.get("имя")); → Шурик
        trace(user.get("age")); → 25
        trace(user.exists("имя")); → true
        trace(user); → { age => 25, имя => Шурик }
    }
}
```

Как видим, здесь даже допустимы ключи на кириллице. Методы ***set*** и ***get*** позволяют устанавливать и извлекать значения полей. Доступна также функция ***exists***, позволяющая проверить наличие заданного ключа в hash. Доступ в точечной нотации здесь невозможен и варианты ***user.age*** или ***user."age"*** здесь не работают.

Implementing Dynamic

Класс ***Dynamic*** можно использовать в собственных классах наподобие интерфейса, то-есть, присоединять с помощью слова ***implements***. При этом объекты класса несколько изменяют свои свойства. Поясним на примере:

```
class Name {
    static function main() {
    var p = new User();
    p.name = "Валя";
    p.age = 12;
    p.a = "Москва";
```

```
        p.a = 0.25;
        p.a = {};
        p.a.x = 777;
        trace(p.a); →  { x => 777 }
        }
}
class User implements Dynamic  {
        public function new() { }
        public var name :String;
        public var age :Int;
}
```

Объект ***p*** имеет тип ***User***, а поля ***name*** и ***age***, объявленные в классе, имеют обычные свойства. Новизна в том, что теперь для объекта ***p*** можно создавать новые динамические поля, не объявленные в классе. Так поле ***p.a*** имеет тип ***Dynamic***, а это значит, что его можно инициировать любыми типами и даже для него можно создавать собственные поля, например ***p.a.x***, как мы делали это ранее.

В классах с ***implements Dynamic*** допустимо указание параметра типа. В этом варианте поля приобретают некоторые дополнительные ограничения. Например:

```
class A implements Dynamic<String> {
        public var x:Int;
        public function new() {}
}
class Prob {
        static public function main() {
                var c = new A();
                c.x = 1;
                c.y = "foo";
                //c.z = 1; → здесь будет ошибка
        }
```

}

Поле ***x*** получает тип ***Int*** и это значит, что объявленные в классе поля могут получать любой тип. Не объявленное поле ***y*** получает тип ***String***, указанный, как параметр типа, и никакой другой тип для ***y*** недопустим. Поэтому не объявленное поле ***z*** не может быть типа ***Int***. То-есть, объявленные и не объявленные поля теперь имеют разные свойства.

Классы с ***implements Dynamic*** (независимо от наличия параметра типа) могут использовать специальную функцию ***resolve*** (которую в классе надо создать). Эта функция используется при вызове не объявленных в классе полей (вызывается автоматически), а в качестве результата она возвращает имя не объявленного поля:

```
class A implements Dynamic {
	public var x:Int;
	public function new() {}
	function resolve(y:String) {
		return 'Проверка функции resolve: $y';
	}
}
class Prob {
	static public function main() {
		var c = new A();
		c.x = 2;
		trace(c.x); → 2
		trace(c.sss); → Проверка функции resolve: sss
	}
}
```

Не очень ясно, где это может быть полезно.

8. Typedef, Enum и ещё кое-что

Typedef

Typedef полезная вещь, он может заменить небольшие классы, а иногда и интерфейсы. Эта структура также создаёт типы, как классы и интерфейсы. Применять ***typedef*** выгодно там, где нужен полиморфизм. Рассмотрим пример:

```
class Prob  {
	public static function main() {
		var f = function(r:Som):Int { return r.g(r.a, r.b); }
		var t = {a:5, b:6, g:function(a:Int, b:Int):Int { return a - b; }}
		trace(f(t));
		var p = new Foo(10, 5);
		trace(f(p));
	}
}
class Foo  {
	public var a:Int;
	public var b:Int;
	public var g:Int->Int->Int;
	public function new(a:Int, b:Int) {
		this.a = a;
		this.b = b;
		g = function(c:Int, d:Int):Int { return a + b + c + d; }
	}
}
typedef Som = { var a:Int; var b:Int; var g:Int->Int->Int; }
```

Значит, после служебного слова ***typedef*** следует имя структуры, а потом после знака равенства объявляются поля, разделённые точкой с запятой и расположенные в фигурных скобках. Для всех полей обязательно должен быть указан тип.

Здесь полиморфизм работает внутри функции ***f***. Её аргументы типизированы как тип ***Som***, определённый через ***typedef***. В примере

использованы два варианта. В первом варианте аргументы функции ***f*** представлены полями анонимного объекта ***t***, а во втором - полями экземпляра класса ***Foo***.

Хотя ***typedef*** не может содержать определение функции, он может иметь ссылку на функцию с определённой сигнатурой. Как раз это и демонстрируется в нашем примере. Чтобы анонимный объект ***t*** и класс ***Foo*** были приняты, как тип ***Some***, в них надо определить метод ***g*** с той же сигнатурой ***:Int -> Int -> Int,*** что и в ***typedef***. Таким образом, чтобы ***typedef***, анонимная структура ***t*** и экземпляр класса ***p*** были взаимозаменяемы, достаточно, чтобы в них были поля с одинаковыми именами и свойствами.

При вызове метода ***g*** экземпляра класса ***p*** имеют место равенства : ***c = a*** и ***d = b***, так как значения ***a*** и ***b*** передаются через конструктор, а переменные ***c*** и ***d*** получают эти же значения, как аргументы метода.

Структуры ***typedef*** могут наследовать друг другу, для чего применяется символ ***(>)***. Вот как это можно использовать:

```
class Prob  {
      static function main()  {
            var t:Point3 = {x :5, y:7, z:9};
            trace(t.z);  → 9
      }
}
typedef Point = { x : Int, y : Int }
typedef Point3 = { > Point, z : Int }
```

Point представляет точку на плоскости, а с помощью механизма наследования мы создали ***Point3*** для точки в пространстве.

Typedef могут иметь необязательные (optional) поля со знаком вопроса впереди:

```
class Prob  {
      static function main()  {
            var s:User = {age :25, name:"Natasha"};
```

```
            trace(s); ->  { name => Natasha, age => 25 }
            trace(s.phone); -> null
        }
}
typedef User = { age : Int, name : String, ?phone : String }
```

Если необязательное поле не инициировано, оно получает значение ***null***, а trace это значение в hash не выводит на терминал.

Поля в typedef могут быть объявлены с применением слова ***var*** и тогда говорят, что это классовая нотация для typedef. Для этого варианта применяется другой синтаксис при наличии optional поля:

```
typedef User = { var age : Int;  var name : String; @:optional var
phone : String; }
```

То-есть, необязательный аргумент отмечается знаком **@.** Этот знак в Haxe применяется для метаданных (metadata), далее мы с ними ещё встретимся.

Часто ***typedef*** применяют для создания псевдонимов (alias), позволяющих заменять громоздкие названия на короткие. Например если создать такие псевдонимы:

```
typedef L = Lambda;
typedef N = neko.Lib;
typedef S = List<String>;
```

то программировать создание списка и вывод на терминал можно так:

```
var a :S = L.list(["aaa", "bbb"]);
trace(a); → {aaa, bbb}
N.println(a); → {aaa, bbb}
```

Заменить на псевдоним можно имя любого объекта, да ещё и вместе с указателем его типа. Особенно часто громоздкие выражения встречаются в web-программировании и там создание псевдонимов может быть более полезным.

По умолчанию typedef и его поля являются ***public*** (в противоположность полям класса) и потому может быть использован

из любого места. Пусть, например, в файле ***Foo.hx*** имеется такой typedef:

```
typedef User = { name :String, age :Int }
```

Тогда его можно использовать, применив ***import***:

```
import Foo;
class A  {
        public static function main()  {
                var u :User;
        }
}
```

Или без ***import***, применив полный путь:

```
class A  {
        public static function main()  {
                var u :Foo.User;
        }
}
```

Но typedef может быть и со спецификатором ***private***:

```
private typedef User  = { name :String, age :Int }
```

и тогда он будет доступен только из того модуля, где был объявлен. Объявлять ***private*** можно и отдельные поля typedef.

Вообще-то typedef в классовой нотации мало отличается от классов и от анонимных объектов, что можно проиллюстрировать на таком примере:

```
class Prob1  {
        public static function main() {
                var v1 :P = { x : 33, y : 55 };
                var v2 = { x : 33, y : 55 };
                trace('$v1 аналогично $v2');  →
  { x => 33, y => 55 } аналогично { x => 33, y => 55 }
                $type(v1);  →  P
                $type(v2);  →  { x : Int; y : Int; }
```

```
    }
}
typedef P = { var x : Int; var y : Int; }
```

Здесь ***v1*** – typedef, а ***v2*** – анонимный объект. Как видим, всё одинаково, отличается только тип, который для ***v1*** будет ***P***, а для ***v2*** - ***{ x : Int; y : Int; }***. По существу это одно и то же.

Итераторы

В Нахе итераторами (***iterable***) называются все те объекты, которые могут быть использованы в циклах ***for***. Например, массивы и ранги являются итераторами. Нахе позволяет создавать свои собственные итераторы. Имеется встроенный typedef с названием ***Iterator***, полями которого являются две функции: ***hasNext*** и ***next***. Вот как выглядит этот ***Iterator***:

```
typedef Iterator = { function hasNext():Bool; function next():T; }
```

Результат, возвращаемый функцией ***next*** является переменной цикла (её тип задан абстрактным и, значит, может быть конкретизирован разными типами), а ***hasNext*** проверяет условие окончания цикла. Пока ***hasNext*** возвращает ***true***, цикл будет продолжаться.

Итератором в Нахе будет любой объект, имеющий методы ***hasNex***t и ***next***. Например, создадим объект ***r***, который будет итератором и представлять последовательность натуральных чисел от начального значения ***(r1)*** до конечного ***(r2)*** через заданный интервал ***(h)***:

```
class Main  {
    static function main() {
        var r = new RevRang(4, 10, 2);
        for (x in r) { trace(x); } → 6 8 10
    }
}
class RevRang {
```

```
    var r2 :Int;
    var i :Int;
    var h :Int;
    public function new(r1 :Int, r2 :Int, h :Int) {
        this.i = r1;
        this.r2 = r2;
        this.h = h;
    }
    public function hasNext() return i != r2;
    public function next() return i+= h;
}
```

В классе ***RevRang*** определены эти два нужных нам метода. Недостаток нашей программы в том, что конечное значение ***r2*** должно удовлетворять условию ***r2 = r1 + n * h***, где ***n*** – целое. Предлагаю самостоятельно устранить этот недостаток.

Шаг можно задать отрицательным и тогда числа будут убывать. Например, зададим такие параметры:

var r = new RevRang(25, 5, -5);

Тогда результат будет: ***20 15 10 5***.

Перечислитель (Enum)

Как и во многих других языках программирования, в Нахе можно использовать перечислитель (Enum), позволяющий выбрать нужное значение из многих. Можно считать, что это разновидность коллекций, но в отличие от массивов и списков, enum создаёт тип. Элементами (полями) ***enum*** могут быть только названия (идентификаторы). Применяется такой синтаксис: служебное слово ***enum***, его название с большой буквы (это тип) и в фигурных скобках поля через точку с запятой. Enum не является итератором и не может быть использован в цикле ***for***, но его можно применять в переключателе ***switch-case***, например:

```
class Foo  {
        static function main()  {
        trace(f(green)); →  зелёный
        trace(f(red)); →  красный
        trace(f(two)); →  это не цвет
        trace(Color.red); →  red
        }
        static function f( c : Color ) : String  {
                return switch( c ) {
                        case red: "красный";
                        case green: "зелёный";
                        case blue: "синий";
                        default: "это не цвет";
                }
        }
}
enum Color {
        red;
        green;
        blue;
        one;
        two;
}
```

Поле enum можно вызвать, применив точечную нотацию:

```
trace(Color.red);
```

Но, как видим, при этом возвращается сам этот идентификатор.

В отличие от других языков, поля enum имеют дополнительные возможности. В частности, они могут иметь аргументы и потому в документации их называют конструкторами. Хотя, эти конструкторы не могут иметь тело а потому не совсем функции. Аргументы

конструкторов перечисляются в круглых скобках через запятую с указанием типа, например:

```
class Prob  {
        static function main()  {
                trace(Color.grey(2)); → grey(2)
                trace(Color.alpha(7, rgb(1, 2, 3)));  → alpha(7,rgb(1,2,3))
                trace(Color.func(g(3))); → func(9)
        }
        public static function g(y :Int) {return y * y; }
}
enum Color { red; green; blue;
        grey( v :Int);
        rgb( r :Int, g :Int, b :Int);
        alpha( a :Int, col :Color );
        func ( f :Dynamic);
}
```

У конструктора ***alpha*** в списке аргументов имеется представитель самого enum ***Color***, то-есть, допускается и рекурсивный вызов. Если объявить аргумент с типом ***Dynamic***, то этот конструктор может принимать функции, экземпляры классов и вообще все допустимые типы. В нашем примере аргументу ***f*** конструктора ***func*** передаётся функция ***g***.

Можно поступать и так:

```
class Prob  {
        static function main()  {
                var s = Color.func(g(7)); → 49
                trace(s); → func(End)
        }
        public static function g(y) { trace(y * y); return "End"; }
}
enum Color {
```

```
    func ( f :Dynamic);
}
```

Есть ограничение: функция ***g*** не может быть ***Void***.

Пока, правда, не очень ясно, велика ли польза, которую можно извлечь из всего этого.

Абстрактные классы

(При первом знакомстве можно пропустить; всё, что выполняется в следующих примерах можно сделать без использования абстрактных классов)

Нахе позволяет использовать абстрактные классы. При объявлении такого класса вместо слова ***class*** используется ***abstract***, а после имени класса обязательно указывается тип в круглых скобках. Конструктор такого класса имеет некоторую особенность. Посмотрим на примере:

```
class Prob {
    static function main() {
        var p = new A('Привет!');
        trace(p); → Привет!
    }
}
abstract A(Dynamic) {
    public function new(x) { this = x; }
}
```

Указанный здесь тип ***A(Dynamic)*** позволяет передавать конструктору значение любого типа. Переменная экземпляра (у нас ***x***) предварительно не объявляется и обязательно должно быть ***this = x;*** (не ***this.x = x***;). Экземпляр класса ***p*** на самом деле теперь представляет переменную экземпляра (то, что обозначено, как ***this***). Других полей и методов абстрактный класс не может иметь. То-есть, вся функциональность должна содержаться только в конструкторе. Например, можно запрограммировать:

```
class Prob {
        static function main() {
                var a = new A(7);
                trace(a); → 49
        }
}
abstract A(Any) {
        public function new(x) { this = x * x; }
}
```

Поведение не изменится, если вместо ***A(Dynamic)*** указать ***A(Any)***. Можно указать конкретный тип, например ***A(String)*** и тогда конструктор будет принимать только значения типа ***String***. В остальном абстрактный класс похож на обычный.

Абстрактные классы используют для создания инструмента по неявному преобразованию типов. При объявлении класса при этом применяют служебные слова ***from — to*** с помощью которых указывают, какие типы преобразуются. Посмотрим на такой пример:

```
class Prob {
        static public function main() {
                var a :A = "false";
                var b  = a;
                trace(b); → false
        }
}
abstract A(Dynamic) from String to Bool {
        function new(x) { this = x; }
}
```

Здесь тип String в выражении ***var b = a;*** неявно приводится к типу Bool, как это было указано в словах ***from String to Bool.*** Тип, указанный в круглых скобках для класса должен быть совместим с обоими преобразуемыми типами (***Dynamic*** совместим с любыми

типами). Если здесь указать, например ***Int***, то возможным будет только преобразование из ***Int*** к ***Float*** (но не наоборот). Почему-то тип ***Any*** здесь не допустим.

Есть и другой способ использования ***from — to*** с применением уже знакомого нам знака **@,** применяемого для метаданных.

```
abstract A(Float) {
    function new(x :Float) { this = x; }
    @:from static function f(s :String) {
        return new A(Std.parseFloat(s));
    }
    @:to function g() { return [this * 3];}
}
class Prob {
    static public function main() {
        var a :A = "3.08";
        var b :Array<Float> = a;
        trace(b); → [9.24]
    }
}
```

Здесь выполняется более сложное преобразование: переменная типа ***String*** преобразуется в массив с элементами типа ***Float***. Интересно, что функции ***f*** и ***g***, выполняющие преобразование, явно даже не упоминаются. Тут имеется целый ряд трюков, понять которые не так легко.

Metadata ***@:op*** позволяет программировать изменение (перегрузку, overloading) методов, в том числе и встроенных операторов, бинарных и одинарных. Например:

```
abstract A(String) {
    public function new(s:String) { this = s + ", "; }
    @:op(T * S) public function f(x :Int) {
        var s = new StringBuf();
```

```
            for (i in 0...x) s.add(this);
            return new A(s.toString() + "много раз");
        }
}
class Prob {
        static public function main() {
            var a = new A("Ещё раз");
            trace(a * 3); →   Ещё раз, Ещё раз, Ещё раз, много раз,
        }
}
```

Здесь бинарный оператор умножения ***(*)*** overload к повторению заданного текста требуемое количество раз. На самом деле с помощью встроенной функции ***StringBuf*** создаётся буфер, в который в цикле ***for*** добавляется текст несколько раз. Функция ***toString*** приводит результат к типу ***String***. (Почему в конце текста выводится ещё одна запятая?)

Reflect (отражение)

В классе ***Reflect*** есть набор методов, позволяющих использовать вместо переменных и методов ссылки на них (отражения). Например, метод ***setProperty*** позволяет в экземпляре любого класса создать новый метод. При этом в классе достаточно только объявить переменную (точнее создать только её идентификатор), а в головной программе можно сделать её методом экземпляра. Например:

```
class Prob {
        static public function main() {
            var p = new A();
            Reflect.setProperty( p, "f", function (x, y) {return x + y;} );
            trace(p.f(4, 7));  → 11
        }
}
```

```
class A  {
        public function new() { };
        public var f :Dynamic;
}
```

Метод ***setProperty*** принимает три аргумента: название созданного перед этим экземпляра класса, имя метода в виде значения типа ***String*** и анонимной функции, определяющей функциональность метода. После этого метод экземпляра можно использовать, как обычно. В примере переменная ***f*** объявлена с типом ***Dynamic*** и тем самым не накладывается никаких ограничений на сигнатуру метода. Но сигнатуру можно и указать и тогда её уже нельзя будет изменить:

```
public var f :Int->Int->Int;
```

Переменную в классе можно объявить ***static*** и тогда не нужно создавать экземпляр, а значит не нужен и конструктор:

```
class Prob {
        static public function main() {
                Reflect.setProperty( A, "f", function (x, y) {return x + y;} );
                trace(A.f(4, 7)); → 11
        }
}
class A  { public static var f; }
```

Здесь даже можно не указывать тип переменной ***f***.

Для методов из класса ***Reflect*** переменные и функции экземпляров классов вызываются с помощью идентификаторов представленных в форме строки. Это открывает некоторые дополнительные возможности:

```
class Prob {
        static public function main():Void {
                var p = new A();
                var x = Reflect.field(p, "h");
                trace(x);
```

```
        Sys.print("Что надо выполнить f1 или f2? ");
        var r :String = Sys.stdin().readLine();
        var g = Reflect.getProperty(p,r);
        trace(g(8));
    }
}
class A {
    public function new() { }
    function f1(t) { return t + t; };
    function f2(t) { return t * t; }
    var h = 7;
}
```

Для доступа к полю применяется метод ***field*** из класса ***Reflect***. При этом имя поля указано, как ***"h"***. Для доступа к методу применяется, соответственно, метод ***getProperty***. Поскольку имя метода тоже указывается, как строка, то можно, например, ввести это имя с клавиатуры на этапе runtime. Если ввести с клавиатуры ***f1***, получим результат ***16***, а если — ***f2*** получим ***64***. Таким образом, появляется возможность во время выполнения программы управлять ею с клавиатуры.

Методы класса ***Reflect*** можно применять и к анонимным объектам:

```
class Prob {
    static public function main() {
        var x = Reflect.field(p, "name");
        trace(x); → Anna
        var g = Reflect.getProperty(p, "f");
        trace(g(0.7)); → 0.644217687237691
    }
    static var p = { name : "Anna", age : 25,
         f : function(x) { return Math.sin(x);} };
```

```
}
```

Не забывать, что анонимный объект должен быть статическим для доступа из статической функции ***main***.

Класс ***Reflect*** имеет ещё такие методы: ***hasField, setField, callMethod, fields, isFunction, compare, compareMethods, isObject, isEnumValue, deleteField, copy.*** О назначении этих методов можно судить по их названиям.

Стек (stack)

Стек (по правилу «последний вошёл, первый вышел») и очередь («первый вошёл, последний вышел») в Нахе можно создавать на основе списка и массива. Кроме того, класс ***GenericStack*** из пакета ***haxe.ds*** позволяет создавать особый итератор, имеющий дополнительные возможности для создания стека. Посмотрим на довольно замысловатом примере:

```
import haxe.ds.GenericStack;
enum E { Land(name:String); EU(ln:Array<E>); }
class Pro {
	static function main() {
		var exit = false;
		var ln = [Land("Россия"), Land("Китай"),
		EU([Land("Италия"), Land("Франция"),
Land("Германия")]),
		Land("Иран"), Land("Индия")];
		var stk = new GenericStack<Iterator<E>>();
		stk.add(ln.iterator());
		do { exit = f(stk); }
		while (!exit);
	}
	public static function f(stk) : Bool {
		if (stk.isEmpty()) { return true; }
```

```
            var x = stk.first();
            if (x.hasNext()) {
            switch (x.next()) {
                  case Land(name): trace("Страна в стеке: " + name);
                  case EU(ln): stk.add(ln.iterator());
                  }
            } else { stk.pop(); }
            return false;
      }
}
```

Для создания стека ***stk*** использован Enum ***E***, имеющий два члена с параметрами:

- ***Land*** с параметром ***name:String***, представляющим название страны,
- ***EU*** с параметром ***ln:Array<E>***, представляющим массив.

Элементами этого массива являются члены ***Land*** этого же Enum, то-есть здесь использован рекурсивный вызов члена Enum. Конкретно параметр члена ***EU*** представляют названия стран, входящих в Европейский союз.

Сначала ***stk*** создаётся, как экземпляр класса ***GenericStack*** с параметром типа ***<Iterator<E>>***. Этот объект пока является пустым списком:

trace(stk); → {}

Затем с мощью метода ***add*** в этот список добавляется итератор, полученный из массива, содержащего элементы Enum ***E***, в результате применения к массиву стандартного метода ***iterator***:

stk.add(ln.iterator());

При выводе на экран получим:

trace(stk); → {{ next => #function:0, a => [Land(Россия),Land(Китай),EU([Land(Италия),Land(Франция),Land(Германия)]),Land(Иран),Land(Индия)], p => 0, hasNext => #function:0 }}

Теперь ***stk*** представляет полноценный стек, с которым можно работать, выполняя некоторые дополнительные манипуляции. Здесь содержатся все элементы массива ***ln*** и необходимые для итератора методы ***next*** и ***hasNext***.

В цикле ***do-while*** с помощью функции ***f*** извлекаются и выводятся на экран все данные, содержащиеся в стеке. Интересно, что условие:
if (stk.isEmpty()) { return true; }
автоматически используется методом ***hasNext*** стека ***stk***. Конечно, разбираться во всех деталях такого поведения нет смысла, можно только этим пользоваться.

9. Работа с текстом

Встроенные методы

Поскольку основное назначение Haxe Web-программирование, то средства обработки текста имеют первостепенное значение. Познакомимся сперва с основными методами из библиотеки Haxe.

В Haxe всё классы и только базовые типы ***Int, Float*** и ***Bool*** являются исключением. Но тип ***String*** – класс, поэтому объявить и инициировать переменную типа ***String*** можно с помощью конструктора класса:
var x = new String("Hello World");
trace(x); → Hello World

Используя краткую форму:
var x = "Hello World";
мы обращаемся к конструктору неявно.

Класс ***String*** имеет широкий набор методов для работы с текстом, некоторые из них аналогичны методам для массивов.

- Метод ***length*** позволяет определить длину текста в символах:
trace(x.length); → 11
- Метод ***charAt*** извлекает из текста отдельный знак по индексу:

trace(x.charAt(6)); → W

- Метод ***charCodeAt*** возвращает цифровой код дл символа:

trace(x.charCodeAt(6)); → 87

- Для одиночного символа цифровой код можно получить с помощью метода ***code***:

trace("W".code); → 87

- Метод ***indexOf*** позволяет узнать индекс указанного символа:

trace(x.indexOf("o")); → 4

Этот метод принимает и второй необязательный аргумент — целое число, указывающее с какого места надо начинать поиск:

trace(x.indexOf("o",5)); → 7

- Метод ***split*** позволяет преобразовать строку в массив. Об этом методе подробно говорилось в разделе «Массивы».
- Метод ***substr*** извлекает из строки подстроку. Он принимает два аргумента: первый указывает начало подстроки, а второй (необязательный) — длину подстроки:

trace(x.substr(3,4)); → lo W

Если второй аргумент не указан, получим подстроку с заданного индекса до конца строки:

trace(x.substr(3)); → lo World

Есть второй вариант этого метода ***substring***, отличающийся тем, что второй аргумент указывает индекс, до которого надо извлечь подстроку (последний символ не входит в подстроку):

trace(x.substring(3,8)); → lo Wo

- Метод ***toLowerCase*** переводит все символы в нижний регистр, а метод ***toUpperCase***, наоборот — в верхний:

trace(x.toLowerCase()); → hello world

trace(x.toUpperCase()); → HELLO WORLD

- Метод ***fromCharCode*** преобразует цифровой код в символ. Почему-то этот метод статический и потому необходимо указывать имя класса:

trace(String.fromCharCode(77)); → M

Переменные типа ***String*** неизменяемы (immutable), то-есть, их нельзя модифицировать «на месте».

Ещё ряд методов для работы с текстом имеется в классе ***StringBuf***:

var x = new StringBuf();

trace(x); → "" - пустая строка.

Конструктор класса ***StringBuf*** не принимает аргументов и создаёт пустой буфер. (Переменная ***x*** получает тип ***StringBuf***). Затем этот буфер может использоваться для работы с текстом. Переменные типа ***StringBuf*** изменяемы (mutable) и в этом их основное отличие от типи String:

- Метод ***add*** позволяет добавлять в буфер текст:

x.add("У лукоморья");

trace(x); → У лукоморья

- Можно использовать метод ***length***:

trace(x.length); → 11

- Метод ***addChar*** добавляет символ, представленный цифровым кодом:

x.addChar(77);

trace(x); → У лукоморьяM – (далее я букву М убрал)

- Метод ***addSub*** позволяет добавить подстроку из заданной строки:

x.addSub("большой дуб",7,4);

trace(x); → У лукоморья дуб

Здесь первый аргумент — заданная строка, второй указывает индекс символа — начала подстроки. Третий аргумент (необязательный) задаёт число символов в подстроке. Если этот аргумент не указать, будет добавлен текст от указанного индекса до конца строки.

- Метод ***toString*** трансформирует переменную к типу ***String***, при этом сама переменная (на месте) не меняет типа:

var y = x.toString();

$type(y); →String

Регулярные выражения

Во всех современных языках основной инструмент обработки текста это регулярные выражения. Синтаксис регулярных выражений в Нахе в основном аналогичен другим языкам, но также есть и некоторые особенности. Регулярные выражения являются объектом и имеют тип ***EReg.*** Они создаются с помощью знака ***(~)*** перед выражением, ограниченном прямыми слешами:

var e:EReg = ~/foo/;

Тип переменной ***e*** можно было бы и не указывать, тогда он был бы выведен из контекста. Поскольку ***EReg*** класс, то регулярное выражение можно также создавать, используя конструктор класса:

var e:Ereg = new e:EReg("foo", "i");

Кроме текста образца для сопоставления, конструктор принимает необязательный аргумент, представленный текстовой константой (в данном случае ***"i"***), задающей флаг (о них несколько позже).

Имеется много встроенных методов для работы с регулярными выражениями. Метод ***match*** позволяет определить, встречается ли заданная комбинация знаков в заданном тексте.

var e = ~/лес/;

var s = "Роняет лес багряный свой убор";

trace(e.match(s)); → true

В Нахе используются стандартные регулярные выражения с применением следующих вспомогательных знаков:

(.) - любой знак

(*) - повторение знака ноль или более раз

(+) - повторение знака один или более раз

[A-Z0-9] – любой символ (буква или цифра), иногда называют рангом

[a-z] – символы могут быть и в нижнем регистре

[^\r\n\t] – управляющие символы

(abc) – заданная группа символов, иногда скобки не обязательны

(^) - начало строки

($) - конец строки

(|) - знак альтернативы ("OR")

После закрывающего прямого слеша может быть флаг, один или несколько через запятую:

var e:EReg = ~/foo/i;

Флаг ***i*** указывает, что регистр знаков в тексте не будет учитываться. Есть ещё такие флаги:

m – позволяет анализировать многострочный текст, его начало должно быть обозначено знаком ***(^)***, а конец знаком ***($)***.

s – Указывает, что точки в тексте надо считать за начало новых строк.

u - Указывает, что используется кодировка UTF-8

Есть ещё флаг ***g***, но о нём надо говорить отдельно.

Посмотрим на следующий пример:

```
class Prob  {
    static function main() {
        var s = "У лукоморья дуб зелёный златая цепь на дубе том";
        var e:EReg = ~/(дуб).+?(цепь).+?(на).+?/;
        if (e.match(s)) {
            trace(e.matched(1)); → дуб
            trace(e.matched(2)); → цепь
            trace(e.matched(3)); → на
            trace(e.matchedLeft()); → У лукоморья
            trace(e.matchedRight()); →дубе том
        }
    }
}
```

Сначала разберём регулярное выражение ***~/(дуб).+?(цепь).+?(на)/***. Это значит, что будут сопоставляться три группы знаков ***дуб, цепь*** и

на. После каждой группы могут быть один или более произвольных, но необязательных знаков.

Если сопоставление удачно, выводятся результаты. Метод ***matched*** вызывается для регулярного выражения e и возвращает результаты только что выполненного сопоставления. Численный аргумент метода определяет, для какой из трёх групп вернуть результат. Если этот аргумент задать равным нулю, будет выведен результат для всех групп совместно вместе с текстом между группами:

trace(e.matched(0)); → дуб зелёный златая цепь на

Метод ***matchedLeft*** возвращает текст до первой группы знаков, а метод ***matchedRight*** – текст от последней группы до конца строки.

Составим теперь программу, отбирающую из текста слова по заданному признаку, например, слова с заглавной буквы:

```
class Prob  {
      static function main() {
            var e:EReg = ~/^[А-Я]/;
            var s = "Павел пел Борис молчал
             Николай ногой качал";
            var m = [];
            for (x in s.split(" ")) {
                  if (e.match(x)) { m.push(x); }
            }
            trace(m); → [Павел,Борис,Николай]
            var r = m.join(" ");
            trace(r); → Павел Борис Николай
      }
}
```

Регулярное выражение ~/^[А-Я]/ означает любая заглавная буква на кириллице, расположенная в начале слова (впрочем, это условие в данном случае необязательно, так как в примере внутри слов

заглавных букв нет и значит знак ^ можно было бы убрать). Для перебора всех слов в цикле мы строку превратили в массив с помощью метода ***split*** по разделителю пробел. При удачном сопоставлении заталкиваем слово в массив, который был создан заранее. Полученный массив затем преобразуем обратно в строку с помощью метода ***join***, а в качестве разделителя опять принимаем пробел.

В примере использован текст из двух строк. Как видим, всё нормально работает и без применения флага ***m***. Впрочем, здесь могут быть всякие нюансы, рассматривать которые подробно у меня нет желания.

Метод ***replace*** позволяет заменять в строке выбранные с помощью регулярного выражения подстроки:

var e:EReg = ~/,.+,/;

var s = "Не жалею, не зову, не плачу";

var r = e.replace(s, ", дорогая Люся,");

trace(r); → Не жалею, дорогая Люся, не плачу

Регулярное выражение ***~/,.+,/*** соответствует произвольному тексту, расположенному между двумя запятыми.

О регулярных выражениях можно написать отдельную книгу. Ограничусь тем, что назову ещё ряд методов класса ***Ereg***:

Метод ***matchedPos() : { pos : Int, len : Int }*** - возвращает позицию (индекс для первого знака) и длину для выбранной подстроки:

trace(e.matchedPos()); → { len => 10, pos => 8 }

Медод ***matchedPos*** не принимает строку, как аргумент, а обращается к той строке, которая была до этого использована другими методами (в нашем случае методом ***replace).***

Метод ***matchSub(s : String, pos : Int, len : Int = -1):Bool***

Проверяет наличие в строке нужной подстроки. Его аргументы — строка, позиция и длина проверяемого участка, при этом третий аргумент можно использовать по умолчанию.

trace(e.matchSub(s,3)); → true

Метод ***map(s : String, f : EReg -> String) : String*** принимает строку и функцию, которая может выполнять над выбранным текстом заданные действия. Результат этих действий подставляется на место выбранного текста.

10. Функциональное программирование

Нахе не является полноценным функциональным языком, но он обладает набором качеств, позволяющих программирование в функциональном стиле. Рассмотрим здесь некоторые из них. Поскольку при функциональном программировании широко применяется pattern matching, рассмотрим сначала дополнительные возможности оператора ***switch-case*** (кроме тех, с которыми познакомились ранее).

Pattern matching

Иногда требуется выбрать какие-то значения из многих возможных. Для того, чтобы не повторять ***case*** много раз, Нахе позволяет примерить приём, показанный в следующем примере:

```
class Prob  {
      public static function main(): Void  {
            for (v in 0...10) {
                  switch v {
                        case x = 3 : trace("x = " + x); → x = 3
                        case x = 7 : trace("x = " + x); → x = 7
                  }
            }
      }
}
```

Здесь также использована локальная переменная, которую можно инициировать после слова ***case***. Тогда справа от знака ***(:)*** эта переменная будет доступна, если сопоставление удачно.

Имеется возможность анализировать много значений одновременно:

```
class Prob  {
      public static function main(): Void  {
            var s = switch [1, false, "foo"] {
                  case [1, false, "bar"]: "0";
                  case [_, true, _]: "1";
                  case [_, false, _]: "2";
                  case _ : "No";
            }
   trace(s); → 2
      }
}
```

Аргумент у ***switch*** представлен массивом, но это не вполне так, поскольку он не рассматривается, как самостоятельная единица и, в частности вариант ***case x : x;*** в этом случае не допустим. Если объявить массив заранее:

```
var m :Dynamic = [1, false, "foo"];
var s = switch m { … }
```

то можно применять и ***case x : x;*** при этом ***x*** будет представлять весь массив, как объект.

Ещё один пример:

```
class Main {
      static function main() {
            switch Std.random(10) {
                  case x = 2 | 4 | 6 : trace("особое число: " + x);
                  case y : trace("другие числа: " + y);
            }
```

```
    }
}
```

Из случайных чисел в диапазоне ***1-10*** отлавливаются числа **2, 4, 6**. Как видим, альтернативные значения образцов в правой по отношению к знаку ***(:)*** части можно перечислять через знак ***(|)***.

Посмотрим ещё на такой пример:

```
class Main {
    static function main() {
        var x = 3;
        switch x {
            case mul(add(_, 1), 5) => res : trace(res); → 20
        }
    }
    static function add(a:Int, b:Int) return a + b;
    static function mul(a:Int, b:Int) return a * b;
}
```

Здесь надо отметить два обстоятельства:

- во-первых, placeholder ***(_)*** распознаётся и на месте аргументов для функций.
- во-вторых, на месте образца для сопоставления после слова case может находится выражение. Это case становится безальтернативным, а в выражение подставляется сопоставляемое значение. Результат вычисления выражения присваивается переменной, которая находится справа от знака ***(=>).***

Вложенные функции можно вызывать последовательно, соединяя шаги тем же знаком ***(=>)***:

```
switch x  {
    case add(_, 1) => mul(_, 5) => res:
    trace(res); → 20
}
```

В следующем примере чётные числа из заданного ранга отмечаются словом ***even***, а нечётные — словом ***odd***.

```
class Main {
	static function main() {
		for (x in 0...5) {
			switch x {
				case f(_) => true : trace(" " + x + ": even");
				case _ : trace(" " + x + ": odd");
			}
		}
	}
	static function f(r) return r % 2 == 0;
}
```

Функция ***f*** возвращает ***true*** для чётных чисел и ***false*** для нечётных (если тело функции представлено одним только оператором ***return***, то фигурные скобки не обязательны). Из примера видим, что если выражение после слова case даёт ***true*** => ***true***, то этот ***case*** имеет удачное сопоставление, а если ***false*** => ***true***, то неудачное.

Получим результат:

```
Main.hx:5:  0: even
Main.hx:6:  1: odd
Main.hx:5:  2: even
Main.hx:6:  3: odd
Main.hx:5:  4: even
```

Знак ***(_)*** позволяет создавать разные варианты образцов при сопоставлении с массивом:

```
class Main {
	static function main() {
		var m = [1, 6, 8];
		switch(m) {
			case [_, 3, 8] : trace("0");
```

```
                case [_, _, 8] : trace("1");
                case [] : trace("2");
                case [_, _, _] : trace("3");
                case _ : trace("4");
            }
        }
}
```

Здесь результат будет ***1***, а если строку ***case [_, _, 8] : trace("1");*** закомментировать, то получим ***3***.

При использовании в pattern matching коллекций, например массивов, открывается много дополнительных возможностей. В следующем примере показан один приём для работы с текстом:

```
class Main {
        static function main() {
            var s = "Роняет лес багряный свой убор";
            switch s.split(" ") {
                case ["Роняет", x, "багряный", "свой", y] :
trace('Выбраны: $x и $y');
                case _ : trace("Неизвестная фраза");
            }
        }
}
```

Заданную фразу сначала преобразуем в массив с помощью метода ***split***. Для того, чтобы выбрать из фразы нужные слова, в качестве образца используем сам этот массив, в котором только эти слова заменяем на переменные. При сопоставлении переменные инициируются элементами массива с теми же индексами. В результате получим фразу: ***Выбраны: лес и убор***.

Посмотрим теперь на пример работы с хешем:

```
class Main {
        static function main() {
```

```
            var person = { name: "Саша", age: 72 };
            switch person {
                case { age: _ > 50 => true} : trace('кто-то старше
50');
                case { name: "Саша", age: 42 } : trace('Это Саша,
ему 42');
                case { name: name } : trace('Это $name');
                case _ : trace("незнакомец");
            }

        }
}
```

Будет получено: ***кто-то старше 50***. Я думаю, тут не требуются комментарии.

Приведу ещё простой пример использования Enum:

```
class Main {
    static function main() {
        var x :Game = WIN(10);
        switch (x) {
            case START : trace('Игра началась');
            case LOST : trace('Вы проиграли');
            case WIN(s) : trace('Вы выиграли со счётом: $s!');
        }

    }
}
enum Game {
    START;
    LOST;
    WIN(s:Int);
}
```

Получим ***Вы выиграли со счётом:10.*** Даже и здесь можно было бы не указывать тип ***x :Game***, тогда он был бы выведен автоматически.

Разнообразные задачи можно решать с использованием регулярных выражений для анализа текста. Например:

```
class Prob {
    static function main() {
        var s = "Жили были дед да баба";
        var m = s.split(" ");
        var e = ~/^б.+$/;
        for (x in m)  {
            switch x  {
                case e.match(x) => true : trace(x);
                case _ : {};
            }
        }
    }
}
```

Здесь мы выбираем из текста слова, начинающиеся с буквы ***б***. В результате получим: ***были баба***.

Рекурсия

Цикл можно создать, применив рекурсию — это когда функция рекурсивно вызывает саму себя. Рекурсия это единственный способ организации циклов в функциональном программировании. Например, для неё можно применить pattern matching:

```
class Prob1 {
    static public function main() {
        trace(f(5));        → 120
    }
    public static function f(x :Int) :Int  {
```

```
        switch x {
            case 0, 1 : return 1;
            default : return x * f(x-1);
        }
    }
}
```

В этом примере вычисляется факториал целого числа.

Рассмотрим ещё простой пример применения рекурсии и pattern matching для создания чистой (без побочных эффектов) функции. Вычислим сумму элементов массива:

```
class Main {
    static function main() {
        var m = [1,2,3,4,5];
        trace(f(m)); → 15
    }
    static function f(m :Array<Int>) :Int  {
        switch m  {
            case [] : return 0;
            case _ : return m.pop() + f(m);
        }
    }
}
```

Метод ***pop*** извлекает последний элемент массива и удаляет его из массива. Таким образом, функция ***f*** вызывается рекурсивно каждый раз для массива с числом элементов на единицу меньше. Выстраивается цепочка: ***5+4+3+2+1+0***. И это типичный пример чистой функции. Можно устранить потребность в большом объёме памяти, если применить «аккумулятор»:

```
class Main {
    static function main() {
        var m = [1,2,3,4,5];
```

```
        trace(f(m, 0)); → 15
    }
    static function f(m :Array<Int>, a :Int) :Int  {
        switch m  {
            case [] : return a;
            case _ : { a = a + m.pop(); return f(m, a); }
        }
    }
}
```

Для функции ***f*** добавляется ещё один аргумент, представляющий аккумулятор ***a***, которому при вызове функции задаётся начальное значение (при суммировании должно быть ***a=0***). Теперь элементы массива суммируются в аккумуляторе.

Нахе обладает и другими качествами, необходимыми для полноценного функционального языка: неизменяемость (immutable) переменных и структур, анонимные функции (closure), строгая статическая типизация, параметры типа и некоторые другие. Например, возможность использования функций в качестве аргументов и возвращаемых результатов для других функций, позволяет применять такую популярную для функциональных языков операцию, как каррирование. Например:

```
class Prob  {
    public static function main(): Void  {
        function g(y) { return f(2,y); }
        trace(g(4)); → 8
        function q(x) { return 5 + x; }
        trace(q(7)); → 12
    }
    public static function f(x,y) {return x * y; }
}
```

Здесь двух-арная функция ***f*** каррируется к одно-арной функцией **g**. Каррировать можно и базовые операторы — функция ***q*** каррирует бинарную операцию сложения.

Очевидно, что можно было бы заняться подробным исследованием языка Haxe на предмет возможности программирования на нём в функциональном стиле.

11.Разное

Использование пакетов

Модули программы можно располагать в пакетах. Текст пакета начинается с директивы ***package***, после которой располагается название пакета, всегда с маленькой буквы. Пакет может содержать один, или несколько классов. В той же директории, где находится головная программа, содержащая класс с функцией ***main***, надо создать папку под названием, совпадающим с именем пакета и поместить в эту папку файл с пакетом. Название файла может быть произвольным и иметь расширение ***hx***. Лучше всего показать технику использования пакета на конкретном примере. Назовём пакет ***mypack*** и, соответственно, создадим папку ***mypack***. Поместим в эту папку файл ***Prob.hx*** со следующим содержанием:

```
package mypack;
class B {
	public var x :Int;
	public function new(x) { this.x = x; }
}
class C {
	public static function double( s:String ):String { return s + s; }
}
```

В той же директории, где находится папка ***mypack***, создадим файл ***Foo.hx*** с головной программой:

```
import mypack.Prob;
class Foo  {
        static function main()  {
        Sys.println(C.double( "Hello! "));  → Hello! Hello!
        var p = new B(777);
        Sys.println(p.x);  → 777
        }
}
```

Как видим, здесь всё работает также, как если бы классы ***B*** и ***C*** располагались в этом же файле ***Foo.hx***. Директива ***import*** загружает пакет.

Вместо директивы ***import*** можно применить ***using*** и всё будет работать также, но при этом ещё появится возможность вызова статического метода ***double*** в точечной нотации:

```
using mypack.Prob;
class Foo  {
        static function main()  {
        Sys.println("World ".double());  → World World
        Sys.println(C.double( "Hello! "));  → Hello! Hello!
        var p = new B(777);
        Sys.println(p.x);  → 777
        }
}
```

При вызове метода ***double*** в точечной нотации даже не требуется указывать имя класса ***C***. (Попробуйте вариант, когда в классах ***B*** и ***C*** имеются методы с одинаковыми именами, например ***double***).

Директива ***import*** позволяет загружать не весь пакет, а только один класс, или даже один только typedef:

```
import mypack.Prob.B;
```

Мультиплатформенность

Основная идея Haxe состоит в том, что исходный код Haxe компилируется (или транслируется) на разные платформы: Neko, JavaScript, PHP, ActionScript3, Java, C++, swf. Это позволяет все части веб-приложения (клиентская часть и серверная логика) разрабатывать на одном языке. Мне трудно судить, насколько эффективен такой подход, можно только отметить, что пока язык не слишком популярен. Но как язык программирования общего назначения Haxe, пожалуй, ничем не уступает большинству современных языков. Нет смысла здесь рассматривать мультиплатформенность языка подробно, тема эта слишком объёмна для начального знакомства с языком. Ограничусь только приведением простейшего примера по созданию JavaScript-кода.

В файле ***Main.hx*** поместим такой код:

```
import js.Browser;
class Main {
   static function main() {
      var button = Browser.document.createButtonElement();
      button.textContent = "Click me!";
      button.onclick = function(event) {
         Browser.alert("Haxe is great");
      }
      Browser.document.body.appendChild(button);
   }
}
```

Здесь использованы несколько методов из класса ***js.Browser***. Назначение этих методов можно понять по их названиям. Файл ***Main.hx*** надо скомпилировать такой командой:

```
haxe -js main-javascript.js -main Main
```

В результате получим файл ***main-javascript.js*** довольно большого объёма по сравнению с исходным файлом. Этот javascript-код можно анализировать, но приводить его здесь нет смысла. Теперь надо создать файл ***index.html***:

```
<!DOCTYPE html>
<html>
   <body>
      <script src="main-javascript.js"></script>
   </body>
</html>
```

Содержимое файла ***main-javascript.js*** следует затем вставить в текст файла ***index.html*** вместо ***"main-javascript.js"***. После этого можно открыть файл ***index.html*** в браузере и тогда увидим страницу с одной только кнопкой ***«Click me!»***. Нажав на эту кнопку, увидим приветствие ***Haxe is great***.

Ясно, что такой примитивный пример — пустая забава, код легче было бы написать сразу на языке JavaScript. Но Haxe имеет много дополнительных возможностей, и, конечно, позволяет избавиться от необходимости вставлять код вручную. Класс js.Browser имеет много различных полезных методов. Однако, пожалуй самое ценное в Haxe — это возможность использования PHP. Но в этом случае даже подобный примитивный пример потребует более сложных действий, а потому ограничимся здесь только этими скудными сведениями по данной теме.

Считается, что освоив язык Haxe можно заниматься веб-программированием без знания, например, языков JavaScript и PHP. Мне кажется, что это преувеличение. Для занятия веб-программированием в любом случае придётся изучить ещё много разнообразный вещей, без которых никак не обойтись.

Я предполагал рассмотреть здесь ещё такие средства языка, как макро-программирование и мета-данные (metadate). Однако, эти темы требуют подробного изложения дополнительного материала, что для первого знакомства с языком вряд ли целесообразно.